Information and Computation 304 (2025) 105296

Contents lists available at ScienceDirect

Information and Computation

journal homepage: www.elsevier.com/locate/yinco

Elastic-degenerate string comparison

Estéban Gabory [a], Moses Njagi Mwaniki [b], Nadia Pisanti [b], Solon P. Pissis [a,c,*], Jakub Radoszewski [d], Michelle Sweering [a], Wiktor Zuba [a,d]

[a] CWI, Amsterdam, the Netherlands
[b] University of Pisa, Pisa, Italy
[c] Vrije Universiteit, Amsterdam, the Netherlands
[d] Institute of Informatics, University of Warsaw, Warsaw, Poland

ARTICLE INFO

Article history:
Received 15 September 2024
Received in revised form 7 March 2025
Accepted 12 March 2025
Available online 14 March 2025

Keywords:
Elastic-degenerate string
Sequence comparison
Languages intersection
Pangenome
Acronym identification

ABSTRACT

An elastic-degenerate (ED) string T is a sequence of n sets $T[1], \ldots, T[n]$ containing m strings in total whose cumulative length is N. We call n, m, and N the length, the cardinality and the size of T, respectively. The language of T is defined as $\mathcal{L}(T) = \{S_1 \cdots S_n : S_i \in T[i] \text{ for all } i \in [1, n]\}$. Given two ED strings, how fast can we check whether the two languages they represent have a nonempty intersection? We call this problem the ED String Intersection (EDSI) problem. For two ED strings T_1 and T_2 of lengths n_1 and n_2, cardinalities m_1 and m_2, and sizes N_1 and N_2, respectively, we show the following:

- There is no $\mathcal{O}((N_1 N_2)^{1-\epsilon})$-time algorithm, for any $\epsilon > 0$, for EDSI even if T_1 and T_2 are over a binary alphabet, unless the Strong Exponential-Time Hypothesis is false.
- There is no combinatorial $\mathcal{O}((N_1 + N_2)^{1.2-\epsilon} f(n_1, n_2))$-time algorithm, for any $\epsilon > 0$ and any function f, for EDSI even if T_1 and T_2 are over a binary alphabet, unless the Boolean Matrix Multiplication conjecture is false.
- An $\mathcal{O}(N_1 \log N_1 \log n_1 + N_2 \log N_2 \log n_2)$-time algorithm for outputting a compact representation of the intersection language of two unary ED strings. When T_1 and T_2 are given in a compact representation, we show that the problem is NP-complete.
- An $\mathcal{O}(N_1 m_2 + N_2 m_1)$-time algorithm for EDSI.
- An $\tilde{\mathcal{O}}(N_1^{\omega-1} n_2 + N_2^{\omega-1} n_1)$-time algorithm for EDSI, where ω is the matrix multiplication exponent; the $\tilde{\mathcal{O}}$ notation suppresses factors that are polylogarithmic in the input size.

© 2025 The Author(s). Published by Elsevier Inc. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

1. Introduction

Sequence (or string) comparison is a fundamental task in computer science, with numerous applications in computational biology [1], signal processing [2], information retrieval [3], file comparison [4], pattern recognition [5], security [6], and elsewhere [7]. Given two or more sequences and a distance function, the task is to compare the sequences in order to infer or visualize their (dis)similarities [8].

* Corresponding author.
E-mail addresses: esteban.gabory@cwi.nl (E. Gabory), njagi.mwaniki@di.unipi.it (M.N. Mwaniki), nadia.pisanti@unipi.it (N. Pisanti), solon.pissis@cwi.nl (S.P. Pissis), jrad@mimuw.edu.pl (J. Radoszewski), michelle.sweering@cwi.nl (M. Sweering), w.zuba@mimuw.edu.pl (W. Zuba).

https://doi.org/10.1016/j.ic.2025.105296
0890-5401/© 2025 The Author(s). Published by Elsevier Inc. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

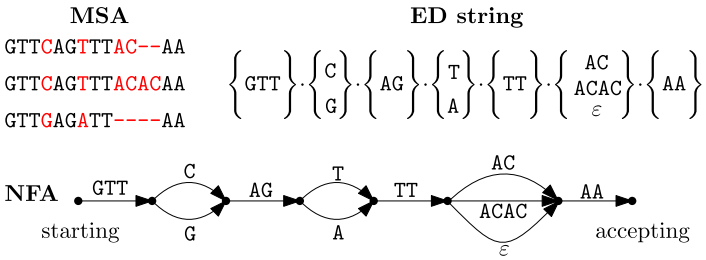


Fig. 1. An example of an MSA and its corresponding (non-unique) ED string T of length $n = 7$, cardinality $m = 11$ and size $N = 20$, and the compacted NFA for T. The compacted NFA can be seen as a special case of an edge-labeled directed acyclic graph.

Many sequence representations have been introduced over the years to account for *unknown* or *uncertain* letters, a phenomenon that often occurs in data that comes from experiments [9]. In the context of computational biology, for example, the IUPAC notation [10] is used to represent loci in a DNA sequence for which several alternative nucleotides are possible as variants. This gives rise to the notion of *degenerate string* (or *indeterminate string*): a sequence of finite sets of *letters* [11]. When all sets are of size 1, we are in the special case of a *standard string* (or *deterministic string*). Degenerate strings can encode the consensus of a population of DNA sequences [12] in a gapless multiple sequence alignment (MSA). Iliopoulos et al. generalized this notion to also encode insertions and deletions (gaps) occurring in MSAs by introducing the notion of *elastic-degenerate string*: a sequence of finite sets of *strings* [13].

The main motivation to consider elastic-degenerate (ED) strings is that they can be used to represent a *pangenome*: a *collection* of closely-related genomic sequences that are meant to be analyzed together [14]. Several other, more powerful, pangenome representations have been proposed in the literature, mostly graph-based ones; see the comprehensive survey by Carletti et al. [15] or by Baaijens et al. [16]. Compared to these more powerful representations, ED strings have algorithmic advantages, as they support: (i) fast and simple on-line string matching [17,18]; (ii) (deterministic) subquadratic string matching [19–21]; and (iii) efficient approximate string matching [22,23].

Our main goal here is to give the first algorithms and lower bounds for comparing two pangenomes represented by two ED strings.[1] We consider the most basic notion of matching, namely, to decide whether two ED strings, each encoding a language, have a nonempty intersection. Like with standard strings, algorithms for pairwise ED string comparison can serve as computational primitives for many analysis tasks (e.g., phylogeny reconstruction); lower bounds for pairwise ED string comparison can serve as meaningful lower bounds for the comparison of more powerful pangenome representations such as, for instance, variation graphs [15].

Let us start with some basic definitions and notation. An *alphabet* Σ is a finite nonempty set of elements called *letters*. By Σ^* we denote the set of all strings over Σ including the *empty string* ε of length 0. For a string $S = S[1]\cdots S[n]$ over Σ, we call $n = |S|$ its *length*. The fragment $S[i\,..\,j]$ of S is an *occurrence* of the underlying *substring* $P = S[i]\cdots S[j]$. We also say that P occurs at *position* i in S. A *prefix* of S is a fragment of S of the form $S[1\,..\,j]$ and a *suffix* of S is a fragment of S of the form $S[i\,..\,n]$. An *elastic-degenerate string* (ED string, in short) T is a sequence $T = T[1]\cdots T[n]$ of n finite sets, where $T[i]$ is a subset of Σ^*. The total size of T is defined as $N = N_\varepsilon + \sum_{i=1}^{n}\sum_{S\in T[i]}|S|$, where N_ε is the total number of empty strings in T. By m we denote the total number of strings in all $T[i]$, i.e., $m = \sum_{i=1}^{n}|T[i]|$. We say that T has *length* $n = |T|$, *cardinality* m and *size* $N = ||T||$. An ED string T can be treated as a compacted nondeterministic finite automaton (NFA) with $n+1$ states, called *segments*, numbered $1,\ldots,n+1$, and m transitions labeled by strings in Σ^*. State 1 is *starting* and state $n+1$ is *accepting*. For each index $i \in [1,n]$ and string $S \in T[i]$, there is a transition from state i to state $i+1$ with label S; inspect also Fig. 1 for an example. The language $\mathcal{L}(T)$ generated by the ED string T is the language accepted by this compacted NFA. That is, $\mathcal{L}(T) = \{S_1\cdots S_n : S_i \in T[i] \text{ for all } i \in [1,n]\}$.

We next define the main problem in scope; inspect also Fig. 2 for an example.

ED String Intersection (EDSI)

Input: Two ED strings, T_1 of length n_1, cardinality m_1 and size N_1, and T_2 of length n_2, cardinality m_2 and size N_2.

Output: YES if $\mathcal{L}(T_1)\cap\mathcal{L}(T_2)\neq\emptyset$, NO otherwise.

Our results We make the following specific contributions:

1. In Section 2.1, we give several conditional lower bounds. In particular, we show that there is no $\mathcal{O}((N_1N_2)^{1-\epsilon})$-time algorithm, thus no $\mathcal{O}\left((N_1m_2+N_2m_1)^{1-\epsilon}\right)$-time algorithm and no $\mathcal{O}\left((N_1n_2+N_2n_1)^{1-\epsilon}\right)$-time algorithm, for any constant $\epsilon > 0$, for EDSI even when T_1 and T_2 are over a binary alphabet, unless the Strong Exponential-Time Hypothesis (SETH) [24,25] or the Orthogonal Vectors (OV) conjecture [26] is false.

[1] Pangenome comparison is one of the central goals of two large EU funded projects on computational pangenomics: PANGAIA (https://www.pangenome.eu/) and ALPACA (https://alpaca-itn.eu/).

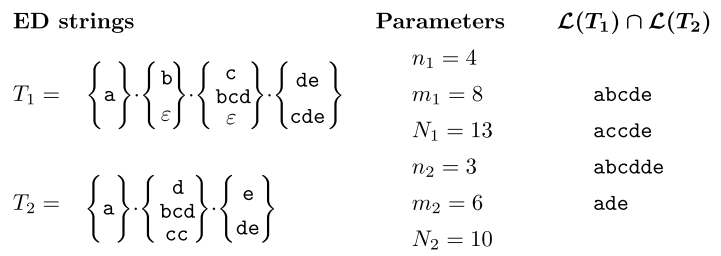


Fig. 2. An example of two ED strings T_1 and T_2 with their parameters and the intersection of their languages. In this instance, we see that $\mathcal{L}(T_1)$ and $\mathcal{L}(T_2)$ have a nonempty intersection.

2. In Section 2.2, we present other conditional lower bounds. In particular, we show that there is no combinatorial[2] $\mathcal{O}((N_1+N_2)^{1.2-\epsilon} f(n_1,n_2))$-time algorithm, for any constant $\epsilon > 0$ and any function f, for EDSI even when T_1 and T_2 are over a binary alphabet, unless the Boolean Matrix Multiplication (BMM) conjecture [27] is false.
3. In Section 3, we show an $\mathcal{O}(N_1 \log N_1 \log n_1 + N_2 \log N_2 \log n_2)$-time algorithm for outputting a compact (RLE) representation of the intersection language of two unary ED strings. In the case when T_1 and T_2 are given in a compact representation, we show that the problem is NP-complete.
4. In Section 4.1, we show an $\mathcal{O}(N_1 m_2 + N_2 m_1)$-time combinatorial algorithm for EDSI in which we assume that the ED strings are over an integer alphabet $[1, (N_1+N_2)^{\mathcal{O}(1)}]$.
5. In Section 4.2, we show an $\tilde{\mathcal{O}}(N_1^{\omega-1} n_2 + N_2^{\omega-1} n_1)$-time algorithm for EDSI, where ω is the matrix multiplication exponent.

Interestingly, we show that the techniques we develop here have applications outside of bioinformatics. Given a sequence $P = P_1, \ldots, P_n$ of n standard strings, we define an *acronym* of P as a string $A = A_1 \cdots A_n$, where A_i is a possibly empty prefix of P_i, for all $i \in [1,n]$. In the Acronym Generation (AG) problem, we are given a dictionary D of k strings of total length K and a sequence P of n strings of total length N, and we are asked to say YES if and only if some acronym of P belongs to D. In Section 5, we show how our techniques for EDSI can be modified to solve AG in $\mathcal{O}(nK+N)$ time.

In Section 6, we show how intersection graphs can be used to solve different ED string comparison tasks. In Section 7, we show how intersection graphs can be used for string matching in the general case when both the pattern and the text are ED strings. In Section 8, we extend our results for EDSI to the approximate case (under the Hamming or edit distance). We conclude this paper in Section 9 with some open problems.

This is a full and extended version of a conference paper [28].

Related work Apart from its applications to pangenome comparison, EDSI is interesting theoretically on its own as a special case of regular expression (regex) matching. Regex is a basic notion in formal languages and automata theory. Regular expressions are commonly used in practical applications to define search patterns. Regex matching and membership testing are widely used as computational primitives in programming languages and text processing utilities (e.g., the widely-used `agrep`). The classic algorithm for solving these problems constructs and simulates an NFA corresponding to the regex, which gives an $\mathcal{O}(MN)$ running time, where M is the length of the pattern and N is the length of the text. Unfortunately, significantly faster solutions are unknown and unlikely [29]. However, much faster algorithms exist for many special cases of the problem: dictionary matching, wildcard matching, subset matching, and the word break problem (see [29] and references therein) as well as for sparse regex matching [30].

Special cases of EDSI have also been studied. First, let us consider the case when both T_1 and T_2 are degenerate strings. In this case, the problem is trivial: EDSI has a positive answer if and only if for every i, $T_1[i] \cap T_2[i]$ is nonempty. Alzamel et al. [31,11] studied the case when T_1 and T_2 are *generalized degenerate strings*: for any $i \in [1,n_1]$ and $j \in [1,n_2]$ all strings in $T_1[i]$ have the same length $\ell_{1,i}$ and all strings in $T_2[j]$ have the same length $\ell_{2,j}$. In the case of generalized degenerate strings, they showed that deciding if $\mathcal{L}(T_1)$ and $\mathcal{L}(T_2)$ have a nonempty intersection can be done in $\mathcal{O}(N_1+N_2)$ time. If T_2 is a standard string, i.e., an ED string with $m_2 = n_2 = 1$, then we can resort to the results of Bernardini et al. [21] for ED string matching. In particular: there is no combinatorial algorithm for EDSI working in $\mathcal{O}(n_1 N_2^{1.5-\epsilon} + N_1)$ time unless the BMM conjecture is false; and we can solve EDSI in $\tilde{\mathcal{O}}(n_1 N_2^{\omega-1} + N_1)$ time. Moreover, Gawrychowski et al. [32] provided a systematic study of the complexity of degenerate string comparison under different notions of matching: Cartesian tree matching; order-preserving matching; and parameterized matching.

Similar to ED strings (and to generalized degenerate strings) is the representation of pangenomes via *founder graphs*. The idea behind founder graphs is that a multiple alignment of few *founder sequences* can be used to approximate the input MSA, with the feature that each row of the MSA is a recombination of the founders. Like founder graphs, ED strings support the recombination of different rows of the MSA between consecutive columns. Unlike ED strings, for which no

[2] The notion of "combinatorial algorithm" is informal but widely used in the literature. Typically, we call an algorithm "combinatorial" if it does not call an oracle for ring matrix multiplication.

efficient index is probable [33] (and indeed their value is to enable fast on-line string matching), some subclasses of founder graphs are indexable, and a recent research line is devoted to constructing and indexing such structures [34–36] as well as investigating the complexity of pattern matching between such special cases of labeled graphs [37]. Indeed, both ED strings and founder graphs are special cases of labeled graphs. Unfortunately, indexing labeled graphs is unlikely to have an efficient solution [38].

2. Conditional lower bounds

In this section, we show several conditional lower bounds for the EDSI problem. Bounds in the first group (see Section 2.1) are based on the popular Strong Exponential-Time Hypothesis (SETH) [39]; the second group of bounds (see Section 2.2) is based, instead, on the Boolean Matrix Multiplication (BMM) conjecture [27].

2.1. Lower bounds based on SETH

We are going to reduce the Orthogonal Vectors (OV, in short) problem to the EDSI problem. In the OV problem we are given two sets $X = \{x_1, \ldots, x_k\}$, $Y = \{y_1, \ldots, y_k\}$ of k binary vectors of length d, and we are asked to decide whether or not there exists a pair $x \in X,\ y \in Y$ of orthogonal vectors; i.e., the dot product of the two vectors is zero. The OV conjecture, implied by SETH (see [26]), is the following.

Conjecture 2.1 *(OV conjecture [26]). The OV problem for two sets of k binary vectors, each of length $d = \Theta(\log k)$, cannot be solved in $\mathcal{O}(k^{2-\epsilon})$ time, for any constant $\epsilon > 0$.*

We show the following reduction.

Theorem 2.2. *Given two sets $X = \{x_1, \ldots, x_k\}$ and $Y = \{y_1, \ldots, y_k\}$ each of k binary vectors of length d, we can construct in linear time two ED strings T_1 and T_2 over a binary alphabet such that:*

- *T_1 has length, cardinality, and size $\Theta(kd)$;*
- *T_2 has length $\Theta(\log k)$, cardinality $\Theta(k)$ and size $\Theta(kd)$; and*
- *There is a pair $(x_a, y_b) \in X \times Y$ of orthogonal vectors if and only if T_1 and T_2 have a nonempty intersection.*

Proof. Let $u_i = 1^d - x_i$ for all $i \in [1\,..\,k]$, where 1^d is the d-dimensional all-1 vector. We construct T_1 and T_2 as follows (see Example 2.3):

$$T_1 = \prod_{i=1}^{k} \prod_{j=1}^{d} \{0, u_i[j]\}, \qquad T_2 = \prod_{i=0}^{\lfloor \log_2 k \rfloor} \left\{0^{d\cdot 2^i}, \varepsilon\right\} \cdot Y \cdot \prod_{i=0}^{\lfloor \log_2 k \rfloor} \left\{0^{d\cdot 2^i}, \varepsilon\right\}.$$

We now show that $\mathcal{L}(T_1)$ and $\mathcal{L}(T_2)$ have a nonempty intersection if and only if there exists a pair of orthogonal vectors $(x_a, y_b) \in X \times Y$.

- Suppose that x_a and y_b are orthogonal. Then for all $j \in [1\,..\,d]$, $y_b[j] \in \{0, u_a[j]\}$ and hence $y_b \in \prod_j \{0, u_a[j]\}$. It follows that

  $$0^{(a-1)d} y_b 0^{(k-a)d} \in 0^{(a-1)d} \prod_j \{0, u_a[j]\}\, 0^{(k-a)d} \subseteq \mathcal{L}(T_1).$$

  By decomposing $a - 1 = \sum_{i \in S_{a-1}} 2^i$ and $k - a = \sum_{i \in S_{k-a}} 2^i$, where for any integer p, the set S_p contains the positions with a 1 in the binary representation of p, we find that

  $$0^{(a-1)d} y_b 0^{(k-a)d} \in \prod_{i \in S_{a-1}} 0^{d\cdot 2^i} \cdot Y \cdot \prod_{i \in S_{k-a}} 0^{d\cdot 2^i} \subseteq \mathcal{L}(T_2).$$

  We conclude that $0^{(a-1)d} y_b 0^{(k-a)d} \in \mathcal{L}(T_1) \cap \mathcal{L}(T_2)$.
- Conversely, suppose that $\mathcal{L}(T_1)$ and $\mathcal{L}(T_2)$ have a nonempty intersection and consider a string $S \in \mathcal{L}(T_1) \cap \mathcal{L}(T_2)$. Let y_b be the vector from Y which is chosen in T_2 when constructing S. The strings in the sets of T_2 all have lengths divisible by d. Thus y_b starts at an index $(a-1)d + 1$ of string S for some integer a. Since $S \in \mathcal{L}(T_1)$, we have $y_b \in \prod_{j=1}^{d} \{0, u_a[j]\}$. This implies that x_a and y_b are orthogonal.

Therefore, solving the orthogonal vectors problem for X, Y is equivalent to checking whether $\mathcal{L}(T_1)$ and $\mathcal{L}(T_2)$ have a nonempty intersection. □

Example 2.3. Let $k = 3$, $d = 3$, $X = \{x_1 = 010,\ x_2 = 100,\ x_3 = 011\}$ and $Y = \{y_1 = 001,\ y_2 = 010\ , y_3 = 110\}$.

We have that

$$T_1 = \left\{ \begin{matrix} 0 \\ 1 \end{matrix} \right\} \cdot \{0\} \cdot \left\{ \begin{matrix} 0 \\ 1 \end{matrix} \right\} \cdot \{0\} \cdot \left\{ \begin{matrix} 0 \\ 1 \end{matrix} \right\} \cdot \left\{ \begin{matrix} 0 \\ 1 \end{matrix} \right\} \cdot \left\{ \begin{matrix} 0 \\ 1 \end{matrix} \right\} \cdot \{0\} \cdot \{0\}$$

and

$$T_2 = \left\{ \begin{matrix} 000 \\ \varepsilon \end{matrix} \right\} \cdot \left\{ \begin{matrix} 000000 \\ \varepsilon \end{matrix} \right\} \cdot \left\{ \begin{matrix} 001 \\ 010 \\ 110 \end{matrix} \right\} \cdot \left\{ \begin{matrix} 000 \\ \varepsilon \end{matrix} \right\} \cdot \left\{ \begin{matrix} 000000 \\ \varepsilon \end{matrix} \right\}.$$

One can observe that each string from $\mathcal{L}(T_1) \cap \mathcal{L}(T_2)$ corresponds to a pair of orthogonal vectors from X and Y. For example, the string 000010000 is in $\mathcal{L}(T_2)$ because $y_2 = 010 \in Y$. Since the vector $x_2 = 100 \in X$ is orthogonal to y_2, one also has $000010000 \in \mathcal{L}(T_1)$. This is because the 3 middle segments of T_1 are constructed to encode any vector which is orthogonal to x_2.

Note that when $d = \Theta(\log k)$, the length n_1, the cardinality m_1 and the size N_1 of T_1 are $\mathcal{O}(k \log k)$, whereas T_2 has length $n_2 = \mathcal{O}(\log k)$, cardinality $m_2 = \mathcal{O}(k)$ and size $N_2 = \mathcal{O}(k \log k)$. Moreover, both ED strings are over a binary alphabet $\Sigma = \{0, 1\}$. This implies various hardness results for EDSI. For example, we can see that, for any constant $\epsilon > 0$, and an alphabet Σ of size at least 2 the problem cannot be solved in

$$\mathcal{O}\left((N_1 + N_2 + n_1 + n_2)^{2-\epsilon} \cdot \text{poly}(n_2)\right)$$

time, conditional on the OV conjecture. By using the fact that $n_1 \leq m_1 \leq N_1$ and $n_2 \leq m_2 \leq N_2$, we obtain the following bounds.

Corollary 2.4. *For any constant $\epsilon > 0$, there exists no*

- $\mathcal{O}((N_1 N_2)^{1-\epsilon})$*-time*
- $\mathcal{O}\left((N_1 m_2 + N_2 m_1)^{1-\epsilon}\right)$*-time*
- $\mathcal{O}\left((N_1 n_2 + N_2 n_1)^{1-\epsilon}\right)$*-time*

algorithm for the EDSI problem, unless the OV conjecture is false.

2.2. Combinatorial lower bounds based on BMM conjecture

In the Triangle Detection (TD, in short) problem we are given three $D \times D$ Boolean matrices A, B, C and one has to check whether there are three indices $i, j, k \in [0, D)$ such that $A[i, j] = B[j, k] = C[k, i] = 1$.[3] It is known that Boolean Matrix Multiplication (BMM) and TD either both have strongly subcubic combinatorial algorithms or none of them do [40]. The BMM conjecture is stated as follows.

Conjecture 2.5 *(BMM conjecture [27]). Given two $D \times D$ Boolean matrices, there is no combinatorial algorithm for BMM working in $\mathcal{O}(D^{3-\epsilon})$ time, for any constant $\epsilon > 0$.*

Our reduction from TD to EDSI is based on the construction of Bernardini et al. from [21] for ED string matching.

Theorem 2.6. *If EDSI over a binary alphabet can be solved in $\mathcal{O}((N_1 + N_2)^{1.2-\epsilon} f(n_1, n_2))$ time, for any constant $\epsilon > 0$ and any function f, then there exists a strongly subcubic combinatorial algorithm for TD.*

Proof. Let D be a positive integer and let A, B, and C be three $D \times D$ Boolean matrices. Further let $s \in [1, D]$ be an integer to be set later. In the rest of the proof, we can assume that s divides D, up to adding α rows and columns containing only 0's to all three matrices, where α is the smallest non-negative representative of the equivalence class $-D \bmod s$.

Let us first construct an ED string $T_1 = X_1 X_2 X_3$ over a large alphabet with $n_1 = 3$, where each X_p, $p \in [1, n_1]$, contains a string for each occurrence of value 1 in A, B and C, respectively. Below i iterates over $[0, D)$, j and k iterate over $[0, \frac{D}{s})$, and x and y iterate over $[0, s)$. Moreover, $x, y \in [0, s)$, v_i for $i \in [0, D)$, and a, \$ are all letters.

- If $A[i, x \cdot \frac{D}{s} + j] = 1$, then X_1 contains the string $v_i x a^j$.

[3] We use this formulation for convenience as our proof is based on another proof which uses this formulation. It is enough to note that when $A = B = C$ is the adjacency matrix of an undirected graph G, this formulation corresponds exactly to solving the TD problem on G.

- If $B[x \cdot \frac{D}{s} + j, y \cdot \frac{D}{s} + k] = 1$, then X_2 contains the string $a^{\frac{D}{s}-j}x\$\$ya^{\frac{D}{s}-k}$.
- If $C[y \cdot \frac{D}{s} + k, i] = 1$, then X_3 contains the string $a^k y v_i$.

The length of each string in each X_p is $\mathcal{O}(D/s)$ and the total number m_1 of strings is up to $3D^2$. Overall, $N_1 = \mathcal{O}(D^3/s)$.

We construct an ED string T_2 with $n_2 = 1$ containing the following strings:

$$P(i, x, y) = v_i x a^{\frac{D}{s}} x\$\$ya^{\frac{D}{s}} y v_i \quad \text{for every } x, y \in [0, s) \text{ and } i \in [0, D).$$

Each string has length $\mathcal{O}(D/s)$ and there are $m_2 = Ds^2$ strings, so $N_2 = \mathcal{O}(D^2 s)$.

We use the following fact.

Fact 2.7 *([21]). $P(i, x, y) \in \mathcal{L}(T_1)$ if and only if the following holds for some $j, k \in [0, D/s)$:*

$$A[i, x \cdot \tfrac{D}{s} + j] = B[x \cdot \tfrac{D}{s} + j, y \cdot \tfrac{D}{s} + k] = C[y \cdot \tfrac{D}{s} + k, i] = 1.$$

We choose $s = \lfloor \sqrt{D} \rfloor$; then $N_1, N_2 = \mathcal{O}(D^{2.5})$ and $n_1, n_2 = \mathcal{O}(1)$. Then indeed an $\mathcal{O}((N_1 + N_2)^{1.2-\epsilon} f(n_1, n_2))$-time algorithm for EDSI would yield an $\mathcal{O}(D^{3-2.5\epsilon})$-time algorithm for the TD problem.

Note also that even though the size of the alphabet used above is $\Theta(s + D) = \Theta(D)$, we can encode all letters by equal-length binary strings blowing N_1 and N_2 up only by a factor of $\Theta(\log D)$ and, hence, obtain the same lower bound for a binary alphabet. □

Both m_1 and m_2 in the reduction are $\mathcal{O}(D^2)$, thus an $\mathcal{O}((m_1 + m_2)^{1.5-\epsilon} f(n_1, n_2))$-time algorithm would yield an $\mathcal{O}(D^{3-2\epsilon})$-time algorithm for TD. We obtain the following.

Corollary 2.8. *If EDSI over a binary alphabet can be solved in $\mathcal{O}((N_1^{1.2} + N_2^{1.2} + m_1^{1.5} + m_2^{1.5})^{1-\epsilon} f(n_1, n_2))$ time, for any constant $\epsilon > 0$ and any function f, then there exists a strongly subcubic combinatorial algorithm for TD.*

3. EDSI: the unary case

An ED string is called *unary* if it is over an alphabet of size 1. In this special case, if both T_1 and T_2 are over the same alphabet $\Sigma = \{a\}$, EDSI boils down to checking whether there exists any $b \geq 0$ such that a^b belongs to both $\mathcal{L}(T_1)$ and $\mathcal{L}(T_2)$.

Let T be a unary ED string of length n over alphabet $\Sigma = \{a\}$. We define the *compact representation* $R(T)$ of T as the following sequence of sets of integers:

$$\forall i \in [1, n]\ R(T)[i] = \{b_{i,1}, b_{i,2}, \ldots, b_{i,m_i}\} \iff T[i] = \{a^{b_{i,1}}, a^{b_{i,2}}, \ldots, a^{b_{i,m_i}}\},$$

where $b_{i,j} \geq 0$ for all $i \in [1, n]$ and $j \in [1, m_i]$, the cardinality of T is $m = \sum_{i=1}^{n} m_i$, and its size is $N = N_\varepsilon + \sum_{i=1}^{n} \sum_{j=1}^{m_i} b_{i,j}$, where N_ε is the total number of empty strings in T.

Theorem 3.1. *If T_1 and T_2 are unary ED strings and each is given in a compact representation, the problem of deciding whether $\mathcal{L}(T_1) \cap \mathcal{L}(T_2)$ is nonempty is NP-complete.*

Proof. The problem is clearly in NP, as it is enough to guess a single element for each set in both T_1 and T_2, and then simply check if the sums match in linear time. We show the NP-hardness through a reduction from the Subset Sum problem, which takes n integers $b_1, b_2, \ldots, b_n$ and an integer c, and asks whether there exist $x_i \in \{0, 1\}$, for all $i \in [1, n]$, such that $\sum_{i=1}^{n} x_i b_i = c$. Subset Sum is NP-complete [41] also for non-negative integers. For any instance of Subset Sum, we set $R(T_1)[i] = \{b_i, 0\}$ (that is, $T_1[i] = \{a^{b_i}, \varepsilon\}$, and ε is the empty string of length 0) for all $i \in [1, n]$, $n_2 = 1$ and $R(T_2)[1] = \{c\}$. Then the answer to the Subset Sum instance is YES if and only if $\mathcal{L}(T_1) \cap \mathcal{L}(T_2)$ is nonempty. □

In what follows, we provide an algorithm which runs in polynomial time in the size of the two unary ED strings when they are given uncompacted.

The set $\mathcal{L}(T)$ can be represented as a set $L(T) \subset \mathbb{N}$ such that $\mathcal{L}(T) = \{a^\ell : \ell \in L(T)\}$. The set $L(T)$ will be stored as a sorted list (without repetitions). We will show how to efficiently compute $L(T_1)$ and $L(T_2)$. Then one can compute $L(T_1) \cap L(T_2)$ in $\mathcal{O}(N_1 + N_2)$ time, which allows, in particular, to check if $L(T_1) \cap L(T_2) = \emptyset$ (which is equivalent to $\mathcal{L}(T_1) \cap \mathcal{L}(T_2) = \emptyset$).

We show the computation for $L(T_1)$. The workhorse is an algorithm from the following Lemma 3.2 that allows to compute the set $L(X_1 X_2)$ of concatenation of two ED strings based on their sets $L(X_1), L(X_2)$.

Lemma 3.2. *Let X_1 and X_2 be ED strings. Given $L(X_1)$ and $L(X_2)$ such that $t_1 = \max L(X_1)$ and $t_2 = \max L(X_2)$, we can compute $L(X_1 X_2)$ in $\mathcal{O}((t_1 + t_2) \log(t_1 + t_2))$ time.*

Proof. For two sets $A, B \subset \mathbb{N}$, by $A + B$ we denote the set $\{a + b : a \in A, b \in B\}$. We then have $L(X_1X_2) = L(X_1) + L(X_2)$. Fast Fourier Transform (FFT) [42] can be used directly to compute $L(X_1) + L(X_2)$ in $\mathcal{O}((t_1 + t_2)\log(t_1 + t_2))$ time. □

Lemma 3.3. *$L(T_1)$ can be computed in $\mathcal{O}(N_1 \log N_1 \log n_1)$ time.*

Proof. We apply the recursive algorithm described in Algorithm 1 to T_1.

Algorithm 1 Compute-$L(T[1]\cdots T[k])$.

```
if k = 1 then
    Compute L(T[1]) naïvely
i ← ⌊k/2⌋
L_1 ←Compute-L(T[1]···T[i])
L_2 ←Compute-L(T[i+1]···T[k])
return L_1 + L_2
```

Let $N_{1,i} = \sum_{x \in L(T_1[i])} \max(1, x)$ and $t_{1,i} = \max L(T_1[i])$ for $i \in [1, n_1]$. Obviously, $t_{1,i} \le N_{1,i}$.

We analyze the complexity of the recursion by levels. For the bottom level, $L(T_1[i])$ can be computed in $\mathcal{O}(N_{1,i})$ time for each $i \in [1, n_1]$, which sums up to $\mathcal{O}(N_1)$. For the remaining levels, we notice that $\max L(T_1[i]\cdots T_1[j]) = t_{1,i} + \cdots + t_{1,j}$. On each level, the fragments of T_1 that are considered are disjoint. Thus, the complexity on each level via Lemma 3.2 is $\mathcal{O}((\sum_{i=1}^{n_1} t_{1,i}) \log(\sum_{i=1}^{n_1} t_{1,i})) = \mathcal{O}(N_1 \log N_1)$. The number of levels of recursion is $\mathcal{O}(\log n_1)$; the complexity follows. □

Theorem 3.4. *If T_1 and T_2 are unary ED strings, then $L(T_1) \cap L(T_2)$ can be computed in $\mathcal{O}(N_1 \log N_1 \log n_1 + N_2 \log N_2 \log n_2)$ time.*

Proof. We use Lemma 3.3 to compute $L(T_1)$ and $L(T_2)$ in the required complexity. Then $L(T_1) \cap L(T_2)$ can be computed via bucket sort. □

4. EDSI: general case

Assuming that the two ED strings, T_1 and T_2, of total size $N_1 + N_2$ are over an integer alphabet $[1, (N_1 + N_2)^{\mathcal{O}(1)}]$, we can sort the suffixes of all strings in $T_1[i]$, for all $i \in [1, n_1]$, and the suffixes of all strings in $T_2[j]$, for all $j \in [1, n_2]$, in $\mathcal{O}(N_1 + N_2)$ time [43].

By $\mathsf{LCP}(X, Y)$ let us denote the length of the longest common prefix of two strings X and Y. Given a string S over an integer alphabet, we can construct a data structure over S in $\mathcal{O}(|S|)$ time, so that when $i, j \in [1, |S|]$ are given to us on-line, we can determine $\mathsf{LCP}(S[i\,..\,|S|], S[j\,..\,|S|])$ in $\mathcal{O}(1)$ time [44].

4.1. Compacted NFA intersection

In this section we show an algorithm for computing a representation of the intersection of the languages of two ED strings using techniques from formal languages and automata theory.

Definition 4.1 *(NFA). A nondeterministic finite automaton* (NFA) is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where Q is a finite set of states; Σ is an alphabet; $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \to \mathcal{P}(Q)$ is a transition function, where $\mathcal{P}(Q)$ is the power set of Q; $q_0 \in Q$ is the starting state; and $F \subseteq Q$ is the set of accepting states.

Using the folklore product automaton construction, one can check whether two NFA have a nonempty intersection in $\mathcal{O}(N_1 \cdot N_2)$ time, where N_1 and N_2 are the sizes of the two NFA [45]. We use a different, compacted representation of automata, which in some special cases allows a more efficient algorithm for computing and representing the intersection.

Definition 4.2 *(Compacted NFA).* An *extended transition* is a transition function of the form $\delta^{ext} : Q \times \Sigma^* \to \mathcal{P}(Q)$, where Q is a finite set of states, Σ^* is the set of strings over alphabet Σ, and $\mathcal{P}(Q)$ is the power set of Q. A *compacted NFA* is an NFA in which we allow extended transitions. Such an NFA can also be represented by a standard (uncompacted) NFA, where each extended transition is subdivided into standard one-letter transitions (and ε-transitions), $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \to \mathcal{P}(Q)$. The states of the compacted NFA are called *explicit*, while the states obtained due to these subdivisions are called *implicit*.

Given a compacted NFA A with V explicit states and E extended transitions, we denote by V^u and E^u the number of states and transitions, respectively, of its uncompacted version A^u. Henceforth we assume that in the given NFA every state is reachable, and hence we have $V^u = \mathcal{O}(E^u)$ and $V = \mathcal{O}(E)$.

Lemma 4.3. *Given two compacted NFA A_1 and A_2, with V_1 and V_2 explicit states and E_1 and E_2 extended transitions, respectively, a compacted NFA representing the intersection of A_1 and A_2 with $\mathcal{O}(V_1^u V_2 + V_1 V_2^u)$ explicit states and $\mathcal{O}(E_1^u E_2 + E_1 E_2^u)$ extended transitions can be computed in $\mathcal{O}(E_1^u E_2 + E_1 E_2^u)$ time if A_1 and A_2 are over an integer alphabet $[1, (E_1^u + E_2^u)^{\mathcal{O}(1)}]$.*

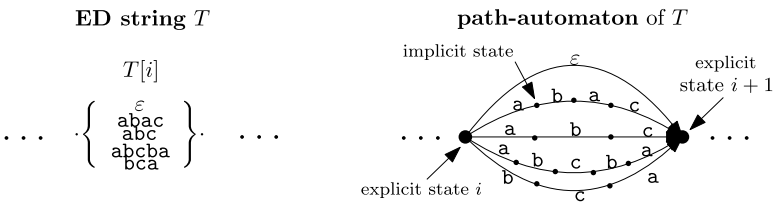


Fig. 3. On the left: an ED string; on the right: the corresponding path-automaton.

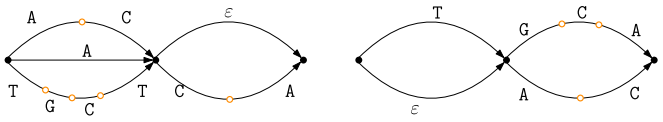


Fig. 4. The path-automata A_1 and A_2 for ED strings $T_1 = \left\{ \begin{matrix} \mathtt{AC} \\ \mathtt{A} \\ \mathtt{TGCT} \end{matrix} \right\} \cdot \left\{ \begin{matrix} \varepsilon \\ \mathtt{CA} \end{matrix} \right\}$ and $T_2 = \left\{ \begin{matrix} \mathtt{T} \\ \varepsilon \end{matrix} \right\} \cdot \left\{ \begin{matrix} \mathtt{GCA} \\ \mathtt{AC} \end{matrix} \right\}$. The filled black nodes are explicit states, while the orange empty nodes are implicit states. (For interpretation of the colors in the figure(s), the reader is referred to the web version of this article.)

Proof. We start by constructing an LCP data structure over the concatenation of all the labels of extended transitions of both NFA of total size $\mathcal{O}(E_1^u + E_2^u)$. It requires $\mathcal{O}(E_1^u + E_2^u)$-time preprocessing and allows answering LCP queries on any two substrings of such labels in $\mathcal{O}(1)$ time.

We construct B, a compacted NFA representing the intersection of A_1 and A_2.

Every state of B is composed of a pair: an explicit state of one automaton and any explicit or implicit state of the other automaton (or equivalently a state of the uncompacted version of the automaton). Thus the total number of explicit states of B is $\mathcal{O}(V_1^u V_2 + V_1 V_2^u)$.

We need to compute the extended transitions of B. For a state (u, v) we check every string pair (P, Q), where P iterates over all extended transitions going out of u and Q iterates over all extended transitions going out of v (a transition going out of an implicit state is represented by a suffix of the transition it belongs to). For every pair (P, Q) we ask an $\mathsf{LCP}(P, Q)$ query. If $\mathsf{LCP}(P, Q)$ is equal to one of $|P|$, $|Q|$ (possibly both), we create an extended transition between (u, v) and the pair of states reachable through those transitions (if one of the transitions is strictly longer, we prune it to the right length, ending it at an implicit state of its input NFA). Otherwise such a transition does not lead to any explicit state of B and thus cannot be used to reach the accepting state; hence we ignore it.

Finally, the starting (resp. accepting) state of B corresponds to a pair of starting (resp. accepting) states of A_1 and A_2.

Since any pair representing an explicit state of B contains an explicit state of A_1 or A_2, the number of such transition pair checks (and hence also the number of the extended transitions of B) is $\mathcal{O}(E_1^u E_2 + E_1 E_2^u)$. Since each such check takes $\mathcal{O}(1)$ time, the construction complexity follows. Note that NFA B may contain unreachable states; such states can be removed afterwards in linear time. The algorithms' correctness follows from the observation that B^u is in fact the standard intersection automaton of A_1^u and A_2^u with some states, that do not belong to any path between the starting and the accepting states, removed. □

We next define the path-automaton of an ED string (inspect Fig. 3 for an example).

Definition 4.4 *(Path-automaton).* Let T be an ED string of length n, cardinality m, and size N. The *path-automaton* of T is the compacted NFA consisting of:

- $V = n + 1$ explicit states, numbered from 1 through $n + 1$. State 1 is the starting state and state $n + 1$ is the accepting state. State $i \in [2, n]$ is the state *in-between* $T[i-1]$ and $T[i]$.
- m_i extended transitions from state i to state $i+1$ labeled with the strings in $T[i]$, for all $i \in [1, n]$, where $E = m = \sum_i m_i$.

The *path-automaton* of T accepts exactly $\mathcal{L}(T)$. The uncompacted version of this path-automaton has $V^u = \mathcal{O}(N)$ states and $E^u = N$ transitions.

An example is constructed in Fig. 4.

Lemma 4.3 thus implies the following result.

Corollary 4.5. *The compacted NFA representing the intersection of two path-automata with $\mathcal{O}(N_1 n_2 + N_2 n_1)$ explicit states and $\mathcal{O}(N_1 m_2 + N_2 m_1)$ extended transitions can be constructed in $\mathcal{O}(N_1 m_2 + N_2 m_1)$ time if the path-automata are over an integer alphabet.*

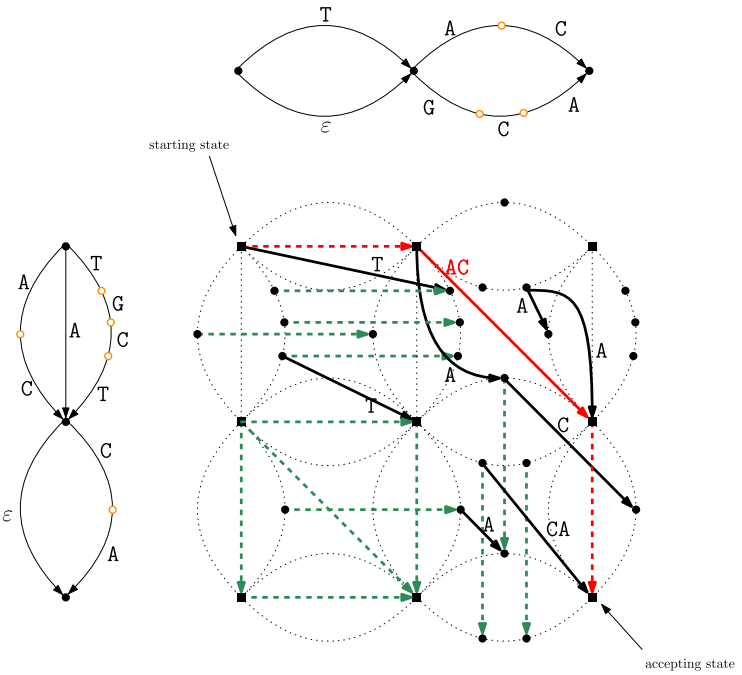


Fig. 5. Intersection automaton for T_1 and T_2 as in Fig. 4 where the string AC in $\mathcal{L}(T_1) \cap \mathcal{L}(T_2)$ that determines a positive answer to the EDSI can be spelled in the path from the starting state to the accepting state. The path-automata A_1 and A_2 are shown on the left and on the top, respectively, and nodes of the intersection automaton are arranged along dotted lines that correspond to copies of the layout of A_1 and A_2, to simplify the understanding of G. The dashed edges of the intersection automata correspond to ε-transitions (namely, transitions such that no letter is read when traversed), while the solid edges correspond to the other extended transitions.

Theorem 4.6. *EDSI for ED strings over an integer alphabet can be solved in $\mathcal{O}(N_1 m_2 + N_2 m_1)$ time. If the answer is YES, a witness can be reported within the same time complexity.*

Proof. The path-automaton of an ED string of size N can be constructed in $\mathcal{O}(N)$ time. Given two ED strings, we can construct their path-automata in linear time and apply Corollary 4.5. By finding any path in the intersection automaton from the starting to the accepting state in linear time (if it exists), we obtain the result. The construction is detailed on an example in Fig. 5. □

Notice that the path-automata representing ED strings, as well as their intersection, are always acyclic, but may contain ε-transitions. In the following we are only interested in the graph underlying the path-automaton, that is the directed acyclic graph (DAG), where every *node* represents an explicit state and every labeled directed *edge* represents an extended transition of the path-automaton (inspect also Fig. 1).

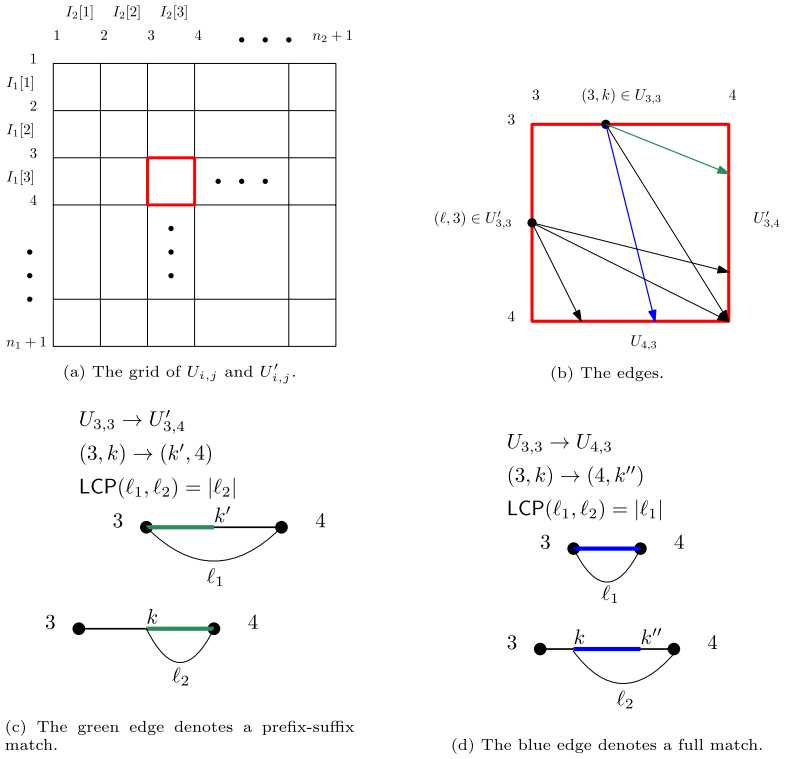
4.2. An $\tilde{\mathcal{O}}(N_1^{\omega-1} n_2 + N_2^{\omega-1} n_1)$-time algorithm for EDSI

In this section, we start by showing a construction of the *intersection graph* computed by means of Lemma 4.3 in the case when the input is a pair of path-automata that allows an easier and more efficient implementation. The construction is then adapted to obtain an $\tilde{\mathcal{O}}(N_1^{\omega-1} n_2 + N_2^{\omega-1} n_1)$-time algorithm for solving the EDSI problem.

For $x \in \{1, 2\}$ by A_x we denote the compacted NFA (henceforth, graph A_x) representing the ED string T_x. By $I_x[i]$ we denote the set of implicit states (henceforth, implicit nodes) appearing on the extended transitions (henceforth, edges) between explicit states (henceforth, explicit nodes) i and $i+1$. For convenience, the implicit nodes in the sets $I_x[1], \ldots, I_x[n_x]$ can be numbered consecutively starting from $n_x + 2$.

Let $U_{i,j} = \{(i, k) : k \in \{j\} \cup I_2[j]\}$ and $U'_{i,j} = \{(k, j) : k \in \{i\} \cup I_1[i]\}$, for all $i \in [1, n_1 + 1]$ and $j \in [1, n_2 + 1]$. As in the construction of Lemma 4.3, the union of all $U_{i,j}$ and $U'_{i,j}$ is the set of explicit nodes of the intersection graph that we construct; this can be represented graphically by a grid, where the horizontal and vertical lines correspond to $U_{i,j}$ and $U'_{i,j}$, respectively (inspect Fig. 6a). In particular, we would like to compute the edges between these explicit nodes (inspect Fig. 6b) in $\mathcal{O}(N_1 m_2 + N_2 m_1)$ time.

Consider an explicit node of the intersection graph; this node is represented by a pair of nodes: one from A_1 and one from A_2. We need to consider two cases: explicit *vs* explicit node; or explicit *vs* implicit node. By symmetry, it suffices to consider an explicit first node. Let us denote this pair by $(i, k) \in U_{i,j}$, where i is an explicit node of A_1 and k is a node of



(a) The grid of $U_{i,j}$ and $U'_{i,j}$.

(b) The edges.

(c) The green edge denotes a prefix-suffix match.

(d) The blue edge denotes a full match.

Fig. 6. An overview of the edges computed by the algorithm.

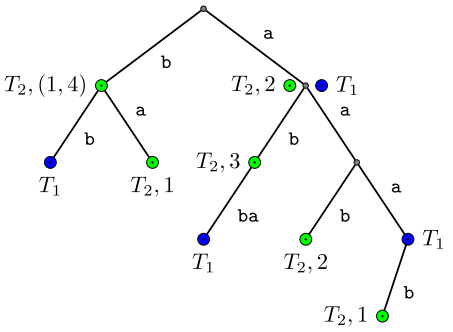


Fig. 7. The annotated compacted trie constructed for $T_1[i] = \left\{ \begin{array}{c} \texttt{abba} \\ \texttt{aaa} \\ \texttt{bb} \\ \texttt{a} \end{array} \right\}$ and $T_2[j] = \left\{ \begin{array}{c} \texttt{ba} \\ \texttt{aaab} \\ \texttt{b} \end{array} \right\}$ in Lemma 4.7. The node corresponding to `b` has two T_2 labels and is an ancestor of the node corresponding to `bb` with a T_1 label; hence two corresponding edges to $U'_{i,j+1}$ are constructed. The node corresponding to `aaa` has a T_1 label and is an ancestor of the node corresponding to `aaab` with a T_2 label; hence a corresponding edge to $U_{i+1,j}$ is constructed. The node corresponding to `a` has both a T_1 and a T_2 label; hence a corresponding edge to $(i+1, j+1)$ is constructed.

A_2. Let us further denote by ℓ_1 the label of one of the edges going from node i to node $i+1$. For k, we have two cases. If k is explicit (i.e., $k = j$) then we denote by ℓ_2 the label of one of the edges going from k to $k+1$. Otherwise (k is implicit), we denote by ℓ_2 the path label (concatenation of labels) from node k to node $j+1$.

As noted in the proof of Lemma 4.3, an edge is constructed only if $\mathsf{LCP}(\ell_1,\ell_2) = \min(|\ell_1|,|\ell_2|)$. If $\mathsf{LCP}(\ell_1,\ell_2) = |\ell_2| < |\ell_1|$ (a prefix of a string in $T_1[i]$ is equal to the suffix of a string in $T_2[j]$ starting at the position corresponding to node $k \in \{j\} \cup I_2[j]$), the edge ends in a node from $U'_{i,j+1}$ (Fig. 6c). If $\mathsf{LCP}(\ell_1,\ell_2) = |\ell_1| < |\ell_2|$ (a whole string from $T_1[i]$ occurs in a string from $T_2[j]$ starting at the position corresponding to node $k \in \{j\} \cup I_2[j]$), the edge ends in a node from $U_{i+1,j}$ (Fig. 6d). Otherwise ($\mathsf{LCP}(\ell_1,\ell_2) = |\ell_1| = |\ell_2|$; the two strings are equal) the edge ends in $(i+1, j+1)$. Symmetrically (i.e., the second node is explicit), the edge going out of a node from $U'_{i,j}$ ends at a node from the set $U'_{i,j+1} \cup U_{i+1,j} \cup \{(i+1, j+1)\}$ (inspect Fig. 6b).

We next show how to construct the intersection graph by computing all such edges going out of $U_{i,j}$ or $U'_{i,j}$ in a *single batch* using suffix trees (inspect Fig. 7 for an example). This construction allows an easier and more efficient implementation

in comparison to the LCP data structure used in the general NFA intersection construction. Let us recall that $||T||$ denotes the size of the ED string T. Henceforth we denote $N_{1,i} = ||T_1[i]||$ for $i \in [1, n_1]$ and $N_{2,j} = ||T_2[j]||$ for $j \in [1, n_2]$.

Lemma 4.7. *For any $i \in [1, n_1 + 1]$ and $j \in [1, n_2 + 1]$, we can construct all $K_{i,j}$ edges going out of nodes in $U_{i,j}$ in $\mathcal{O}(N_{1,i} + N_{2,j} + K_{i,j})$ time using the generalized suffix tree of the strings in $T_2[j]$. We assume that the letters of strings in $T_2[j]$ are over an integer alphabet $[1, N_{2,j}^{\mathcal{O}(1)}]$.*

Proof. Let us start with a simple implementation of the lemma that works *with high probability*. We will then give the details for a deterministic implementation.

We first construct the generalized suffix tree of the strings in $T_2[j]$ in $\mathcal{O}(N_{2,j})$ time [43]. We also mark each node corresponding to a suffix of a string in $T_2[j]$ with a T_2-label. Each such node is also decorated with one or multiple starting positions, respectively, from one or multiple elements of $T_2[j]$ sharing the same suffix. For each branching node of the suffix tree, we construct a hash table, to ensure that any outgoing edge can be retrieved in constant time based on the first letter (the key) of its label. This can be done in $\mathcal{O}(N_{2,j})$ time with perfect hashing [46]. We next spell each string from $T_1[i]$ from the root of the suffix tree making implicit nodes explicit or adding new ones if necessary to create the compacted trie of all those strings; and, finally, we mark the reached nodes of the suffix tree with a T_1-label. Spelling all strings from $T_1[i]$ takes $\mathcal{O}(N_{1,i})$ time.

Every pair of different labels marking two nodes in an ancestor-descendant relationship corresponds to exactly one outgoing edge of the nodes in $U_{i,j}$: (i) if a node marked with a T_2-label is an ancestor of a node marked with a T_1-label, then the suffix of a string from $T_2[j]$ matches a prefix of a string from $T_1[i]$ forming an edge ending in $U'_{i,j+1}$; (ii) if a node marked with a T_1-label is an ancestor of a node marked with a T_2-label, then a string from $T_1[i]$ occurs in a string from $T_2[j]$ extending its prefix and forming an edge ending in $U_{i+1,j}$; (iii) if a node is marked with a T_1-label and with a T_2-label, then the suffix of a string from $T_2[j]$ matches a string from $T_1[i]$ forming an edge ending in $(i + 1, j + 1)$. After constructing the generalized suffix tree of $T_2[j]$ and spelling the strings from $T_1[i]$, it suffices to make a DFS traversal on the annotated tree to output all $K_{i,j}$ such pairs of nodes.

Let us note that the perfect hashing can be avoided under the assumption of the lemma stating that strings in $T_1[i] \cup T_2[j]$ are over an integer alphabet $[1, (N_{1,i} + N_{2,j})^{\mathcal{O}(1)}]$. It suffices to construct the generalized suffix tree of $T_1[i] \cup T_2[j]$. Then one can trim all the nodes of the tree that do not have in their subtree a node corresponding to a (whole) string in $T_1[i]$ or a substring of a string in $T_2[j]$. This makes the construction deterministic. □

Theorem 4.8. *We can construct the intersection graph of T_1 and T_2 in $\mathcal{O}(N_1 m_2 + N_2 m_1)$ time using the suffix tree data structure and tree search traversals if T_1 and T_2 are over an integer alphabet $[1, (N_1 + N_2)^{\mathcal{O}(1)}]$.*

Proof. We will apply Lemma 4.7 for $U_{i,j}$ and $U'_{i,j}$, for all $i \in [1, n_1 + 1]$ and $j \in [1, n_2 + 1]$. To this end, before computing $U_{i,j}$ or $U'_{i,j}$, for each $i \in [1, n_1 + 1]$ and $j \in [1, n_2 + 1]$ we may need to renumber the letters in strings in $T_1[i]$ and $T_2[j]$ with consecutive integers to make sure that they belong to an integer alphabet $[1, (||T_1[i]|| + ||T_2[j]||)^{\mathcal{O}(1)}]$. This can be done in $\mathcal{O}(N_1 n_2 + n_1 N_2)$ total time using one global radix sort.

We have that the total number of nodes is $\sum_{i,j} \mathcal{O}(N_{1,i} + N_{2,j}) = \mathcal{O}(N_1 n_2 + N_2 n_1)$, and then the number of all output edges is bounded by $\mathcal{O}(N_1 m_2 + N_2 m_1)$ by Corollary 4.5. □

Note that if we are interested only in checking whether the intersection is nonempty, and not in the computation of its graph representation, it suffices to check which of the nodes are *reachable* from the starting node, which may be more efficient as there are $\mathcal{O}(N_1 n_2 + N_2 n_1)$ explicit nodes in this graph.

Let X be the set of nodes of $U_{i,j}$ that are reachable from the starting node. From this set of nodes we need to compute two types of edges (inspect Fig. 6b). The first type of edges, namely, the ones from X to $U'_{i,j+1} \cup \{(i+1, j+1)\}$ (green edges in Fig. 6b), are computed by means of Lemma 4.9, which is similar to Lemma 4.7. For the second type of edges, namely, the ones from X to $U_{i+1,j} \cup \{(i + 1, j + 1)\}$ (blue edges in Fig. 6b), we use a reduction to the so-called *active prefixes extension* problem [21] (Lemma 4.11).

Lemma 4.9. *For any given $X \subseteq U_{i,j}$, we can compute the subset of $U'_{i,j+1} \cup \{(i + 1, j + 1)\}$ containing all and only the nodes that are reachable from the nodes of X in $\mathcal{O}(N_{1,i} + N_{2,j})$ time. We assume that the letters of strings in $T_1[i] \cup T_2[j]$ are over an integer alphabet $[1, (N_{1,i} + N_{2,j})^{\mathcal{O}(1)}]$.*

Proof. In Lemma 4.7, the edges from nodes of $U_{i,j}$ to nodes of $U'_{i,j+1}$ come from a pair of nodes in the generalized suffix tree of $T_2[j]$ enriched with strings from $T_1[i]$: one marked with a T_1-label and its ancestor marked with a T_2-label. Notice that the T_2-labels are in a correspondence with the elements of $U_{i,j}$ (the labels on a proper suffix of a string in T_2 are in a one-to-one correspondence with $U_{i,j} \setminus \{(i, j)\}$, and (i, j) corresponds to whole strings in $T_2[j]$), and hence we can trivially remove the T_2-labels that do not correspond to the elements of X. Furthermore, we are not interested in the set of starting positions decorating a node with a T_2-label; we are interested only in whether a node is T_2-labeled or not (i.e., we do

not care from which node of X the edge originates). Since the nodes marked with a T_1-label have in total $N_{1,i}$ ancestors (including duplicates), we can compute the result of this case in $\mathcal{O}(N_{1,i} + N_{2,j})$ time in total. Finally, the node $(i+1, j+1)$ is reachable when a single node is marked with both a T_1-label and a T_2-label. This can be checked within the same time complexity. □

The remaining edges (blue edges in Fig. 6b) are dealt with via a reduction to the following problem.

ACTIVE PREFIXES (AP)

Input: A string P of length m, a bit vector W of size m, and a set $\mathcal{S}$ of strings of total length N.

Output: A bit vector V of size m with $V[p] = 1$ if and only if there exists $P' \in \mathcal{S}$ and $p' \in [1, m]$, such that $P[1\,..\,p'-1] \cdot P' = P[1\,..\,p-1]$ and $W[p'] = 1$.

Bernardini et al. have shown the following result in [21], which relies on fast matrix multiplication (FMM).

Lemma 4.10 *([21]). The AP problem can be solved in $\tilde{\mathcal{O}}(m^{\omega-1}) + \mathcal{O}(N)$ time, where ω is the matrix multiplication exponent.*

Lemma 4.11. *For any given $X \subseteq U_{i,j}$, we can compute the subset of $U_{i+1,j}$ containing all and only the nodes that are reachable from the nodes of X in $\tilde{\mathcal{O}}(N_{1,i} + N_{2,j}^{\omega-1})$ time.*

Proof. The problems of computing the subset of $U_{i+1,j}$ reachable from X and the AP problem can be reduced to one another in linear time.

For the forward reduction, let us set $\mathcal{S} = T_1[i]$ and $P = \prod_{S \in T_2[j]} \$S$, where \$ is a letter outside of the alphabet of T_1. This means that we order the strings in $T_2[j]$ in an arbitrary but fixed way. For a single string $\$S$ (where $S \in T_2[j]$), the positions from $S[1\,..\,|S|-1]$ correspond to the implicit nodes (along the path spelling S) of $I_2[j]$, while the position with \$ corresponds to the explicit node j of A_2. Through this correspondence, we can construct two bit vectors W and V, each of them of size $|P|$, and whose positions are in correspondence with $\{j\} \cup I_2[j]$ (note that this correspondence is not a bijection, as the explicit node j has several preimages when $|T_2[j]| \geq 2$). As $U_{i,j}$ and $U_{i+1,j}$ are in a 1-to-1 correspondence with $\{j\} \cup I_2[j]$, we use the same correspondence to match positions between W and $U_{i,j}$ and between V and $U_{i+1,j}$. Finally, we set $W[k] = 1$ if and only if the corresponding node of $U_{i,j}$ belongs to X. After solving AP, we have $V[k] = 1$ for some k corresponding to a node of $U_{i+1,j}$ if and only if this node is reachable from X. For an example, see Fig. 8.

In more detail, observe that since \$ does not belong to the alphabet of T_1, a string S from $T_1[i]$ has to match a fragment of a string from $T_2[j]$ to set $V[k]$ to 1. This happens only if additionally $W[k - |S|] = 1$; both things happen at the same time exactly when: (i) there exists a node $(i, \ell) \in X$; (ii) there exists an edge from (i, ℓ) to $(i+1, \ell')$; and (iii) the positions $k - |S|$ and k in P correspond to ℓ, ℓ', respectively.

In the above reduction we have $|P| = \sum_{S \in T_2[j]} (|S| + 1) = \mathcal{O}(N_{2,j})$, and $||\mathcal{S}|| = N_{1,j}$, hence the lemma statement follows by Lemma 4.10.

For the reverse reduction, given an instance of AP, we encode it by setting $T_1[i] = \mathcal{S}$, $T_2[j] = \{P\}$ ($N_{1,i} = ||\mathcal{S}||$, $N_{2,j} = |P|$) and X containing the nodes corresponding to positions k where $W[k] = 1$.

This reduction shows that a more efficient solution to the problem of finding the endpoints of edges originating in X would result in a more efficient solution to the AP problem. □

Theorem 4.12. *We can solve EDSI in $\tilde{\mathcal{O}}(N_1^{\omega-1} n_2 + N_2^{\omega-1} n_1)$ time, where ω is the matrix multiplication exponent. If the answer is YES, we can output a witness within the same time complexity.*

Proof. It suffices to set the starting node $(1, 1)$ as reachable, apply Lemmas 4.9 and 4.11, and their symmetric versions for $U'_{i,j}$, for each value of $(i, j) \in [1, n_1 + 1] \times [1, n_2 + 1]$ in lexicographical order, with X equal to the set of reachable nodes of $U_{i,j}$ (respectively of $U'_{i,j}$); and, finally, check whether node $(n_1 + 1, n_2 + 1)$ is set as reachable. Before applying Lemma 4.9, each time we renumber the alphabet in $T_1[i] \cup T_2[j]$ to make it integer. We bound the total time complexity of the algorithm by:

$$\sum_{i,j} \tilde{\mathcal{O}}(N_{1,i}^{\omega-1} + N_{2,j}^{\omega-1}) = \tilde{\mathcal{O}}(n_2 \sum_i N_{1,i}^{\omega-1} + n_1 \sum_j N_{2,j}^{\omega-1}) = \tilde{\mathcal{O}}(N_1^{\omega-1} n_2 + N_2^{\omega-1} n_1).$$

If $\mathcal{L}(T_1) \cap \mathcal{L}(T_2)$ is nonempty, that is, if the node $(n_1 + 1, n_2 + 1)$ is set as reachable from node $(1, 1)$, then we can additionally output a witness of the intersection – a single string from $\mathcal{L}(T_1) \cap \mathcal{L}(T_2)$ – within the same time complexity. To do that we mimic the algorithm on the graph with reversed edges. This time, however, we do not mark all of the reachable nodes; we rather choose a single one that was also reachable from $(1, 1)$ in the forward direction. This way, the marked nodes form a single path from $(1, 1)$ to $(n_1 + 1, n_2 + 1)$. The witness is obtained by reading the labels on the edges of this path. □

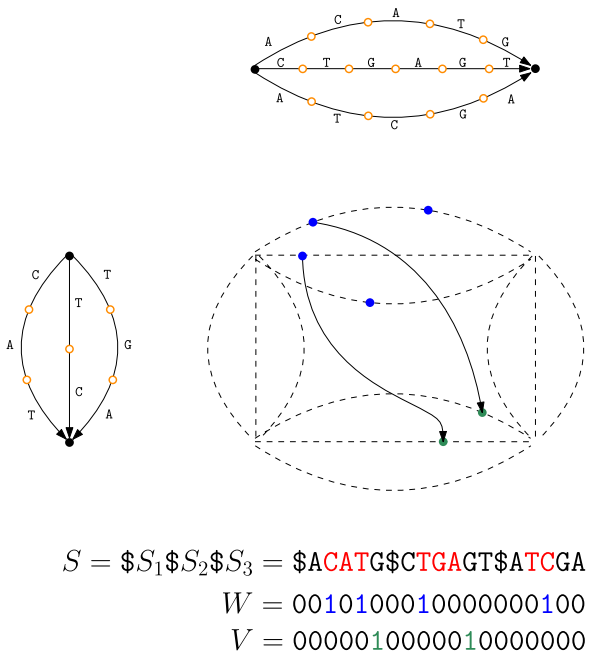


Fig. 8. The path automata for $T_1[i] = \{\mathtt{CAT}, \mathtt{TC}, \mathtt{TGA}\}$ and $T_2[j] = \{\mathtt{ACATG}, \mathtt{CTGAGT}, \mathtt{ATCGA}\} = \{S_1, S_2, S_3\}$, as well as the corresponding part of the intersection graph (top). The blue nodes correspond to a set $X \subseteq U_{i,j}$. Computing the states in $U_{i+1,j}$ that are reachable from X (in green on the figure) can be done by computing the instance of AP with S and W on the bottom part of the figure. The occurrences of strings from $T_1[i]$ in S are highlighted in red.

Observe that if $n_2 = 1$, that is, T_2 is simply a set of standard strings, no node in $U'_{i,1}$ other than $(i, 1) \in U_{i,1}$ is reachable. Indeed, nodes $(i, 1)$ can be reached from $(1, 1)$ through ε-transitions from T_1, but reaching other nodes would require reading a letter, that is, also a change in the state of T_2, and the only explicit state other than 1 in T_2 is 2 (and $(i, 2) \in U_{i,2}$). Due to this, the symmetric version of Lemma 4.11 never needs to be used to compute transitions between $U'_{i,1}$ and $U'_{i+1,2}$. This allows for a more efficient solution in this case.

The same observation improves the time complexity of the algorithms in Theorems 4.6 and 4.8 in the case that $n_2 = 1$. It suffices to consider edges outgoing from nodes (x, y) in the intersection graph such that x is explicit in A_1; the number of such edges is $\mathcal{O}(N_2 m_1)$.

Corollary 4.13. *If $n_2 = 1$ then the running time in Theorems 4.6 and 4.8 is $\mathcal{O}(N_1 + N_2 m_1)$ and the running time in Theorem 4.12 is $\tilde{\mathcal{O}}(N_1 + N_2^{\omega-1} n_1)$.*

This observation will be useful in case of the generalized versions of EDSI showed in Sections 7 and 8, where we can compare the running time of our algorithms with the running time of the existing solutions solving special cases of those EDSI generalizations.

5. Acronym generation

In this section, we study a problem on standard strings. Given a sequence $P = P_1, \ldots, P_n$ of n strings we define an *acronym* of P as a string $A = A_1 \cdots A_n$, where A_i is a (possibly empty) prefix of P_i, $i \in [1, n]$. We next formalize the Acronym Generation problem.

Acronym Generation (AG)

Input: A set D of k strings of total length K and a sequence $P = P_1, \ldots, P_n$ of n strings of total length N.

Output: YES if some acronym of P is an element of D, NO otherwise.

The AG problem is underlying real-world information systems (e.g., see https://acronymify.com/ or https://acronym-generator.com/) and existing approaches rely on brute-force algorithms or heuristics to address different variants of the problem [47–54]. These algorithms usually accept a sequence P of $n \leq n_{\max}$ strings, for some small integer $n_{\max}$, which highlights the lack of efficient exact algorithms for generating acronyms. Here we show an exact polynomial-time algorithm to solve AG for any n.

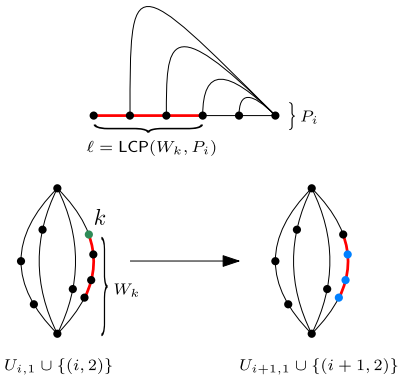


Fig. 9. For the AG problem, the automaton for the prefixes of P_i can be represented as the one on top of the figure. The sets $U_{i,1} \cup \{(i,2)\}$ and $U_{i+1,1} \cup \{(i+1,2)\}$ can be represented as copies of the path-automaton of D, as in the general case. A green node k in $U_{i,1} \cup \{(i,2)\}$ is specified. The red segments form a path reading the LCP string between W_k and P_i. The nodes that are reachable from k in $U_{i+1,1} \cup \{(i+1,2)\}$ are highlighted in blue.

We can encode AG by means of EDSI and modify the developed methods. The AG problem reduces to EDSI in which $T_1[i]$, $i \in [1,n]$, is the set of all prefixes of P_i and $T_2[1] = D$. We have $N_1 \leq N^2$, $m_1 = N$, $n_1 = n$, $N_2 = K$, $m_2 = k$, and $n_2 = 1$, so by Corollary 4.13, we obtain solutions to the AG problem working in $\mathcal{O}(N^2 + KN)$ and $\tilde{\mathcal{O}}(N^2 + K^{\omega-1}n)$ time, respectively.

Since, however, all elements of set $T_1[i]$ are prefixes of a single string (P_i), we can obtain a more efficient graph representation of T_1 by joining nodes i and $i+1$ with a single path labeled with P_i, with an additional ε edge between every (implicit) node of the path and node $i+1$. As the size of the graph for T_1 is smaller ($\mathcal{O}(N)$ nodes and edges), by using Lemma 4.3 we obtain an $\mathcal{O}(kN + KN) = \mathcal{O}(KN)$-time algorithm for solving the AG problem.

The considered ED strings have additional strong properties. $T_1[i]$'s are not just sets of prefixes of single strings, but sets of all their prefixes, while the length n_2 of T_2 is equal to 1. By employing these two properties we obtain the following improved result.

Theorem 5.1. *AG can be solved in $\mathcal{O}(nK + N)$ time.*

Proof. The algorithm of Theorem 4.12 is based on finding out which elements of sets $U_{i,j}, U'_{i,j}$ are reachable. However, since $n_2 = 1$, the sets $U'_{i,j}$ are trivialized. As in Corollary 4.13, it is enough to focus on the transitions between $U_{i,j}$ and $U_{i+1,j} \cup \{(i+1,2)\}$.

In Lemma 4.11, to compute the reachable nodes of $U_{i+1,j}$ knowing the reachable nodes of $U_{i,j}$, fast matrix multiplication is employed (Lemma 4.10), but in this special case a simpler method will be more effective. For $k \neq 1$, let W_k be the unique string read between nodes k and 2 in the path-graph of T_2. The crucial observation is: the edges going out of node $(i,k) \in U_{i,1} \cup \{(i,2)\}$ for $k \neq 1$ end in nodes $(i+1,k')$ for $k' \in [k, k+\ell]$, where $\ell = \mathsf{LCP}(P_i, W_k)$ as the strings from $T_1[i]$ matching the prefix of W_k are exactly all the prefixes of P_i of length at most ℓ (see Fig. 9).

Hence, to compute the reachable subset of $U_{i+1,1} \cup \{(i+1,2)\}$, we can handle the edges going out of $(i,1)$ separately in $\mathcal{O}(K + |P_i|)$ time by letter comparisons, then compute the $\mathsf{LCP}(P_i, W_k)$ values for all the reachable nodes (i,k) either using the LCP data structure, or with the use of the generalized suffix tree of $T_2[1] = D$ in $\mathcal{O}(K + |P_i|)$ total time, and finally, using a 1D line sweep approach, compute the union of the obtained intervals in $\mathcal{O}(K)$ time. In the end, we answer YES if and only if node $(n+1,2)$ is reachable.

Over all values of i this gives an algorithm running in $\sum_i \mathcal{O}(K + |P_i|) = \mathcal{O}(nK + N)$ total time. □

Furthermore, one may be allowed to choose, for each $i \in [1,n]$, a lower bound $x_i \geq 0$ on the length of the prefix of P_i used in the acronym (some strings should not be completely excluded from the acronym). The only modification to the algorithm in such a generalization is replacing intervals $[k, k+\ell]$ by $[k+x_i, k+\ell]$, which does not influence the claimed complexity.

Corollary 5.2. *If the answer to the instance of the AG problem is YES, we can output all strings in D which are acronyms of P within $\mathcal{O}(nK + N)$ time.*

Proof. We want to find out which strings of T_2 represent acronyms of P. In the algorithm employed by Theorem 5.1 the reachable nodes of $U_{i,1} \cup \{(i+1,2)\}$ are found. If there exists an edge from a reachable node (i,k) for $k \notin \{1,2\}$ to $(i+1,2)$ for some $i \in [1,n_1]$, then the path of the path-graph of T_2 containing node k represents an acronym of P. If node $(i+1,2)$ is reached directly from reachable node $(i,1)$, then the whole prefix of P_i used to do that is in D, and hence is a standalone acronym of P. If for a path neither of the two cases qualifies, then it cannot be used to reach node $(n+1,2)$, and hence is not an acronym of P. □

If the generalization with minimal lengths of prefixes is applied, then the values of i used here are restricted to $[i', n]$, where i' is the largest value of i with a restriction $x_i > 0$: node $(i'-1, 2)$ does not have an edge to node $(i', 2)$, and hence does not belong to any path from $(1, 1)$ to $(n+1, 2)$.

Let us remark that although the main focus of real-world acronym generation systems is on the natural language parsing and interpretation of acronyms, our new algorithmic solution may inspire practical improvements in such systems or further algorithmic work.

6. ED string comparison tasks

In this section, we show some applications of our techniques from Section 4. In particular, we show how intersection graphs can be used to solve different ED string comparison tasks.

We consider two ED strings, T_1 of length n_1, cardinality m_1 and size N_1, and T_2 of length n_2, cardinality m_2 and size N_2. We call *intersection graph* the underlying graph of an automaton representing $\mathcal{L}(T_1) \cap \mathcal{L}(T_2)$. By Corollary 4.5 such an automaton (and therefore the corresponding intersection graph) can be constructed in $\mathcal{O}(N_1 m_2 + N_2 m_1)$ time. In Section 4, such a graph was used to check whether $\mathcal{L}(T_1) \cap \mathcal{L}(T_2)$ is nonempty.

In the following, we present other applications of intersection graphs (computed by means of Corollary 4.5 or Lemma 4.7) to tackle several natural ED string comparison tasks with no additional time complexity. We always assume that T_1 and T_2 are over an integer alphabet $[1, (N_1 + N_2)^{\mathcal{O}(1)}]$.

6.1. Shortest/longest witness

Let us start with the most basic application.

EDSI Shortest/Longest Witness
Input: Two ED strings, T_1 of length n_1, cardinality m_1 and size N_1, and T_2 of length n_2, cardinality m_2 and size N_2.
Output: A shortest (resp. longest) element of $\mathcal{L}(T_1) \cap \mathcal{L}(T_2)$ if it is a nonempty set, FAIL otherwise.

Fact 6.1. *The* EDSI Shortest/Longest Witness *problem can be solved in $\mathcal{O}(N_1 m_2 + N_2 m_1)$ time by using an intersection graph of T_1 and T_2.*

Proof. We compute the intersection graph G as the underlying graph of the automaton computed in Corollary 4.5. Given an edge (k, k') in G, we assign a weight $w(k, k')$ equal to the length of its string label (the string ε has length 0). Note that, by construction, G is a directed acyclic graph (DAG). Thus the problem reduces to computing the shortest or the longest path between a source and a sink of a DAG, a problem with a well-known linear-time solution that involves topological sorting. By reading the labels on the shortest (resp. longest) path in $\mathcal{O}(N_1 + N_2)$ time, we can output the shortest (resp. longest) element of $\mathcal{L}(T_1) \cap \mathcal{L}(T_2)$.[4] □

6.2. Counting pairs of matching strings

In the next task, we would like to compute the total number of matching pairs of strings in $\mathcal{L}(T_1) \times \mathcal{L}(T_2)$ considering multiplicities in $\mathcal{L}(T_1)$ and $\mathcal{L}(T_2)$. We assume that the multiplicity of a string S in $\mathcal{L}(T_1)$, which we denote by $\mu(T_1, S)$, is the number of sequences $S_1 \in T_1[1], \ldots, S_{n_1} \in T_1[n_1]$ such that $S_1 \cdots S_{n_1} = S$. The definition for T_2 is analogous.

EDSI Matching Pairs Count
Input: Two ED strings, T_1 of length n_1, cardinality m_1 and size N_1, and T_2 of length n_2, cardinality m_2 and size N_2.
Output: $\sum_{S \in \mathcal{L}(T_1) \cap \mathcal{L}(T_2)} \mu(T_1, S) \cdot \mu(T_2, S)$.

Each matching pair can be represented by a pair of *alignments*: the sequence of $n_1 + n_2$ choices of a single production $S_i \in T_1[i]$ and $S'_j \in T_2[j]$ for each $i \in [1, n_1]$ and $j \in [1, n_2]$, such that the resulting standard string is the same. Such a pair of alignments can in turn be represented by a path in the intersection graph of T_1 and T_2 as the edges correspond to (parts of) productions in some $T_1[i]$ and in some $T_2[j]$.

The representation is almost unique; the only reason this does not need to be the case is when a node $(i+1, j+1)$ is reached from a node (i, j) for $i \in [1, n_1]$, $j \in [1, n_2]$ using only ε-edges. In this case, the three subpaths $(i, j) \to (i+1, j+1)$, $(i, j) \to (i, j+1) \to (i+1, j+1)$ and $(i, j) \to (i+1, j) \to (i+1, j+1)$, all correspond to the choice of productions $\varepsilon \in T_1[i]$ and $\varepsilon \in T_2[j]$, even though this choice should be counted as one.

[4] In the case of a shortest witness, we can obtain an equally efficient algorithm by employing Dijkstra's algorithm using bucket queue since the bound $\mathcal{O}(N_1 + N_2)$ on the weight of the path is known beforehand.

To fix the notation, we call an ε-edge: *vertical*, when it leads from a node (i,k) to $(i+1,k)$ for some explicit node i of A_1; *horizontal*, when it leads from a node (k,j) to $(k,j+1)$ for some explicit node j of A_2; and *diagonal* if it leads from a node (i,j) to $(i+1,j+1)$, where both i and j are explicit nodes.

The problem becomes even more complicated when a few such ε-productions are used in a row in both ED strings (a node $(i+x, j+y)$ is reached from (i,j) using only ε-edges), as a single alignment would correspond to all the "down, right or diagonal" paths in the $x \times y$ grid. The number of such paths can be large, and even if we remove all such diagonal ε-edges (those can be always simulated with a single horizontal and a single vertical one), we cannot remove the horizontal or vertical ones, as those can be traversed by other paths independently. We are still left with $\binom{x+y}{x}$ equivalent subpaths from (i,j) to $(i+x, j+y)$. In order to mitigate this problem we will restrict the usage of such subpaths of many ε-edges to a single, regular one.

We call a path *ε-regular*, if it does not use diagonal ε-edges, and if two ε-edges are used sequentially, then they either have the same direction, or the second one is horizontal. For example, if $T_1 = \{\mathtt{AC}\} \cdot \left\{ \begin{matrix} \mathtt{A} \\ \varepsilon \end{matrix} \right\} \cdot \{\mathtt{GT}\}$ and $T_2 = \left\{ \begin{matrix} \mathtt{A} \\ \mathtt{G} \end{matrix} \right\} \cdot \{\mathtt{C}\} \cdot \left\{ \begin{matrix} \mathtt{G} \\ \varepsilon \end{matrix} \right\} \cdot \{\mathtt{GT}\}$, the matching pair count is 1, for the string ACGT that has multiplicity one in both ED strings. In the intersection graph, this unique alignment corresponds to three different paths: one traversing the diagonal ε-edge, one traversing first the horizontal and then the vertical ε-edge, and finally, the unique corresponding ε-regular path traversing the vertical ε-edge first and then the horizontal one.

Lemma 6.2. *There is a one-to-one correspondence between pairs of alignments that produce the same string and ε-regular paths from the starting to the accepting node in the intersection graph of T_1 and T_2.*

Proof. Consider a pair of alignments; it can be represented by the produced string together with some positions in-between letters marked with (n_1+1) T_1-labels and (n_2+1) T_2-labels in total specifying the beginning and ending of the productions used in $T_1[i]$ and $T_2[j]$, for $i \in [1, n_1]$, $j \in [1, n_2]$. A single position can be marked with both types of label and even many labels of the same type (when ε-productions are used). Each position corresponds to a pair of states in the path-automata: those pairs of states with at least one label read from left to right form a path in the intersection graph, as the T_i-label shows that the state is explicit in T_i. If two labels of a different type appear in the same place, this corresponds to a node composed of two explicit states. When a single position contains multiple labels from both types, the exact ordering between those labels represents the actual path in the graph. At the same time from the point of view of T_1 and T_2 separately all those orderings correspond to exactly the same pair of alignments. This is where ε-regularity plays its role and only one subpath is created (all T_1-labels are placed before the T_2-labels). This way we have defined an injection from the pairs of alignments to the paths of the intersection graph (each pair of alignments corresponds to only one ε-regular path).

On the other hand the labels of the edges in the path represent (the parts of) the transitions used in each automaton, and hence also a pair of alignments, thus the function is also surjective.

We have thus shown that the function relating a pair of alignments to a path is both injective and surjective, and hence a bijection (the one-to-one correspondence from the statement). □

The main result then follows.

Fact 6.3. *The* EDSI Matching Pairs Count *problem can be solved in $\mathcal{O}(N_1 m_2 + N_2 m_1)$ time by using an intersection graph of T_1 and T_2.*

Proof. We compute the intersection graph as the underlying graph of the automaton computed in Corollary 4.5 or Lemma 4.7. By Lemma 6.2, we are counting distinct ε-regular paths from the starting node to the accepting one. As previously, we can sort the nodes topologically. If we wanted to count all the paths it would be enough to compute for each node k the value $S(k)$ equal to the number of paths from the starting node to k. One would have $S(k) = \sum_{\{(k',k)\} \in E(G)} S(k')$, over all edges (k',k) (including parallel edges with different labels) and setting $S((1,1)) = 1$ where $(1,1)$ is the starting node. This time, however, we apply slightly more complicated formulas, which count the paths ending with the ε-edges separately from the other ones, and among those separately depending on the direction of the ε-edge – by the definition of ε-regular paths we do not use any vertical ε-edge directly after a horizontal one, and do not use diagonal ε-edges at all. For a node k, let $S(k)$, $S^h(k)$ and $S^v(k)$ denote the number of ε-regular paths from the starting node to the node k that end with an edge with a non-empty label, with a horizontal ε-edge and with a vertical ε-edge, respectively.

- $S(k) = \sum_{\{(k',k) \text{ positive length edge}\}} S(k') + S^h(k') + S^v(k')$
- $S^h(k) = \sum_{\{(k',k) \text{ horizontal } \varepsilon\text{-edge}\}} S(k') + S^h(k') + S^v(k')$
- $S^v(k) = \sum_{\{(k',k) \text{ vertical } \varepsilon\text{-edge}\}} S(k') + S^v(k')$

By induction, for the accepting node (n_1+1, n_2+1), the number of distinct ε-regular paths from the starting to the accepting node is equal to the sum of the values S, S^h and S^v computed for node (n_1+1, n_2+1). □

6.3. ED matching statistics

Asking whether $\mathcal{L}(T_1) \cap \mathcal{L}(T_2)$ is not nonempty, as a way to tell if two ED strings have something in common, can be too restrictive in practical applications. We will thus consider two more elaborate ED string comparison tasks that consider local matches rather than a match that necessarily involves the entire ED strings from beginning to end. Both notions that we consider, Matching Statistics and Longest Common Substring, are heavily employed on standard strings for practical applications, especially in bioinformatics.

We start by extending the classic Matching Statistics problem [1] from the standard string setting to the ED string setting. Although the solution presented below has already been described in [55], we provide it also here as an intermediate step of the solution to the next comparison task.

ED Matching Statistics

Input: Two ED strings, T_1 of length n_1, cardinality m_1 and size N_1, and T_2 of length n_2, cardinality m_2 and size N_2.

Output: For each $i \in [1, n_1]$, the length $\mathsf{MS}[i]$ of the longest prefix of a string in $\mathcal{L}(T_1[i\,..\,n_1])$ that is a substring of a string in $\mathcal{L}(T_2)$.

Fact 6.4 *([55]). The* ED Matching Statistics *problem can be solved in $\mathcal{O}(N_1 m_2 + N_2 m_1)$ time by using an intersection graph of T_1 and T_2.*

Proof. This time we will use a slightly augmented version of the intersection graph coming from the unpruned intersection automaton. Namely, we construct the automaton as in Corollary 4.5, but do not remove the unreachable parts or the "partial transitions". That is, when we process an explicit state corresponding to a pair (u, v) of states in the path-automata A_1 and A_2, and a pair (s, t) of transitions going out of u and v, we construct the corresponding transition to the pair of states that can be reached through s and t, even if this transition finishes in a pair of implicit states, namely when $0 < \mathsf{LCP}(\ell_1, \ell_2) < \min(|\ell_1|, |\ell_2|)$, where ℓ_1 and ℓ_2 are the respective labels of s and t. Even in that case, the number of transition pair checks remains the same, and therefore the total size of the constructed underlying graph G stays $\mathcal{O}(N_1 m_2 + N_2 m_1)$.

Once again we assign to each edge the weight w storing the length of its string label and process the nodes in the reversed topological order to compute for each node k the value $M(k)$ equal to the length of the longest path from k in A_1 that matches a path in A_2; we have $M(k) = \max_{k'} M(k') + w(k, k')$ where k' iterates over all successors of k. One has $M(k) = 0$ for the nodes that do not have successors (for example the accepting node or nodes corresponding to a pair of implicit states).

By construction, we have $M((i, v)) = \ell$, for an explicit state i of A_1 and a state v of A_2, if and only if ℓ is equal to the maximal LCP between a pair (S_1, S_2), where S_1 a string in $\mathcal{L}(T_1[i\,..\,n])$ and S_2 is a string read starting at (explicit or implicit) state v in A_2.

For every explicit state i of A_1 we can compute $\mathsf{MS}[i] = \max_v M((i, v))$ over all (explicit or implicit) states v of A_2 to obtain the output. □

An example of this construction is shown in Fig. 10.

6.4. ED longest common substring

We now proceed to extending the classic Longest Common Substring problem [1] from the standard string setting to the ED string setting.

ED Longest Common Substring

Input: Two ED strings, T_1 of length n_1, cardinality m_1 and size N_1, and T_2 of length n_2, cardinality m_2 and size N_2.

Output: A longest string that occurs in a string of $\mathcal{L}(T_1)$ and a string of $\mathcal{L}(T_2)$

In the ED Matching Statistics problem only strings starting in explicit nodes of A_1 were considered; here we want to lift this restriction. Computing the value of M for every node of A_1 against every node of A_2 would take $\Theta(N_1 N_2)$ time. We are only interested in the globally maximal value of M. Hence, we can focus only on the computation of those values for a certain subset of those pairs.

Fact 6.5. *The* ED Longest Common Substring *problem can be solved in $\mathcal{O}(N_1 m_2 + N_2 m_1)$ time by using an intersection graph of T_1 and T_2.*

Proof. We start from the augmented graph G, as defined in the proof of Fact 6.4, and computed in $\mathcal{O}(N_1 m_2 + N_2 m_1)$ time. In addition to the standard edges between the explicit nodes, the graph contains edges from a pair containing at least one explicit state to a pair of implicit states, that are inclusion-wise maximal, that is, cannot be extended to obtain a longer

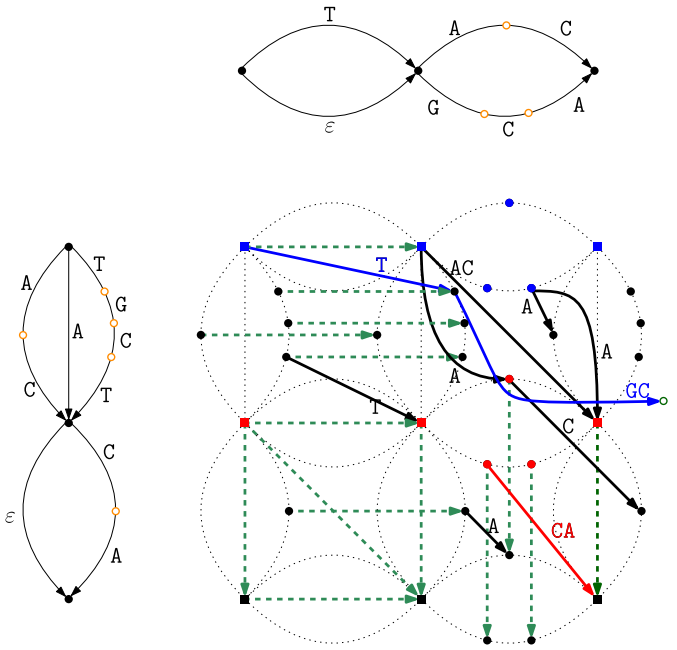


Fig. 10. Matching statistics on the intersection graph of T_1 and T_2 as defined in Fig. 4. To simplify the understanding, we also draw A_1 and A_2 on the left and on the top, respectively. Note that this time, the pairs of implicit nodes that are reachable in a single extended transition from one pair that was previously computed are added. In the figure, there is only one such extra node that we represent by a green empty circle at the right of the graph. Here we highlight paths that are relevant for computing the Matching Statistics. In order to compute MS[1], we look at the paths starting at nodes (i, j) where i is the explicit state 1 in the path-automaton of T_1, and return the length of the longest label of such a path. These are the paths starting in one of the blue nodes (these are the nodes that correspond to the uppermost explicit node of A_1 paired with any node of A_2, that is, they correspond to the uppermost dotted copy of A_2); the longest of such paths (also drawn in blue) corresponds to the string TGC having length 3, and therefore, MS[1] = 3. For MS[2] we do the same, but using as starting nodes those in red that correspond to the internal explicit node of A_1 paired with any node of A_2 (that is, the nodes of the middle dotted copy of A_2). Here the longest path is drawn in red and it spells the string CA, and therefore we set MS[2] = 2.

edge. For the ED Longest Common Substring problem, we additionally need edges symmetric to those – the (inclusion-wise maximal) edges starting in pairs of implicit states that end in pairs containing at least one explicit state. By a symmetric argument (argument for the reversed automata), there are $\mathcal{O}(N_1m_2 + N_2m_1)$ such edges and all can be computed together with the rest of the graph within the same time complexity. We compute the values $M(k)$ for each node k as previously, and find their global maximum. Notice that if $M(k)$ for an implicit node k associates k with another implicit node k', we do not need to compute the value $M(k)$. Indeed, then (k, k') lies in the middle of an edge and the first node of its predecessor on the edge always has a greater value of M.

We are left with the nodes that are not (weakly) connected to any explicit node. The isolated edges containing such nodes were not computed at all. Those edges correspond to strings that, in both ED strings, are fully contained in a single set. Hence, it is enough to compute the longest common substring of two strings, one being a concatenation of strings in $\bigcup_i T_1[i]\#$ and the other a concatenation of strings in $\bigcup_j T_2[j]\$$, for sentinel letters #, $. This can be done in $\mathcal{O}(N_1 + N_2)$ time.

The method of obtaining the witness is the same as in the previous problems (after computing both endpoints of the optimal path using the algorithm described above). □

6.5. ED longest common subsequence

Finally, we show how to extend the classic Longest Common Subsequence (LCS) problem [1] from the standard string setting to the ED string setting. We remark that, unlike the previous problems, in the standard string setting, LCS is not solvable in linear time: there exists a conditional lower bound saying that no algorithm with running time $\mathcal{O}(n^{2-\epsilon})$, for any $\epsilon > 0$, can solve LCS on two length-n strings unless SETH fails [56].

ED Longest Common Subsequence
Input: Two ED strings, T_1 of size N_1, and T_2 of size N_2.
Output: A longest string that occurs in a string of $\mathcal{L}(T_1)$ and a string of $\mathcal{L}(T_2)$ as a subsequence.

Fact 6.6. *The* ED Longest Common Subsequence *problem can be solved in* $\mathcal{O}(N_1 \cdot N_2)$ *time by using an uncompacted intersection graph of* T_1 *and* T_2.

Proof. Consider the uncompacted path-automata for T_1 and T_2, having respective sizes $\mathcal{O}(N_1)$, $\mathcal{O}(N_2)$. For every single transition in those graphs (that is a single letter transition since they are uncompacted), we add a parallel ε-transition (that is, we allow to skip the letter). The sizes of the automata remain $\mathcal{O}(N_1)$ and $\mathcal{O}(N_2)$, and the intersection has size $\mathcal{O}(N_1 \cdot N_2)$. We can then find the longest witness (Fact 6.1) of this intersection in time linear in the size of the automaton, that is, in $\mathcal{O}(N_1 \cdot N_2)$ time. □

Notice that when T_1 and T_2 are standard strings, then $n_1 = m_1 = n_2 = m_2 = 1$ and $\mathcal{O}(N_1 \cdot N_2)$ running time matches the conditional lower bound for standard strings up to subpolynomial factors. Hence, no N_i can be replaced by m_i or n_i. In particular, an $\mathcal{O}(N_1 m_2 + N_2 m_1)$ time algorithm would refute the conditional lower bound.

Proposition 6.7. *In all the solutions presented in this section (as well as in the previous ones) the algorithm can be slightly modified to achieve space complexity $\mathcal{O}(N_1 + N_2)$ without any increase in the running time.*

Proof. Notice that all the problems are solved through computing a certain value (reachability, $L(k), S(k), M((i,v))$) for all the explicit nodes of the intersection graph in a topological order. Each time this order is compatible with the order of sets of T_1 and T_2, and the recurrent formulas refer only to nodes from a single set $U_{i,j}$ or $U'_{i,j}$, that is, the basic computation is enclosed in a single cell of the grid (inspect Fig. 6a). For a single cell of the grid the space used for computation is $\Theta(N_{1,i} + N_{2,j})$, while globally the information passed between the cells is bounded by $\mathcal{O}(N_1 + N_2)$ by going through the cells (i,j) in a lexicographical (or reversed lexicographical) order (it is enough to store information for nodes of $U_{i,j}, U'_{i,j}$ only for the next two values of i). Finally the size of the input as well as of all the generalized suffix trees is also bounded by $\mathcal{O}(N_1 + N_2)$. □

7. (Doubly) elastic-degenerate string matching

String matching (or pattern matching) on standard strings can be seen as a natural (and very useful) generalization of string equality. Since EDSI is basically a counterpart of the string equality problem (we seek for a pair of standard strings from $\mathcal{L}(T_1)$ and from $\mathcal{L}(T_2)$ that are equal), the problem of generalizing it to string matching arises naturally [13]. In particular, the ED String Matching (EDSM) problem of locating the occurrences of a pattern that is a standard string in an ED text has already been considered [19–21,17,13].

For a standard string p and an ED text T, one wants to check whether p occurs in T, that is, if it is a substring of some string $t \in \mathcal{L}(T)$ (decision version), or find all the segments i of T such that an occurrence of p starts within a string of the set $T[i]$; i.e., there exists a standard string $t \in \mathcal{L}(T)$ with an alignment such that p occurs in t starting at a position lying in the ith segment of the alignment.

In this section, we consider the EDSM problem in its full generality, i.e., both the text and the pattern are ED strings. We then show how our solution instantiates to the special cases, where one of the two ED strings is a standard string.

Let us now define the general problem in scope:

DOUBLY ED STRING MATCHING

Input: An ED string T (called *text*) and an ED string P (called *pattern*).

Output: YES if and only if there is a string $p \in \mathcal{L}(P)$ that is a substring of a string $t \in \mathcal{L}(T)$ (decision version); or all segments i such that there is a string $p \in \mathcal{L}(P)$ whose occurrence in T starts in $T[i]$ (reporting version).

Through applying the solution to reversed ED strings P and T (we reverse the sequence of segments and the strings inside them), we can see that the reporting version can be used to find all the segments i where an occurrence ends. Further note that DOUBLY ED STRING MATCHING is equivalent to asking for the union of the results of EDSM over all the patterns from $\mathcal{L}(P)$ – we take a binary OR of the results for the decision version.

The algorithms underlying Theorem 4.6 and Theorem 4.12 can both be extended to solve DOUBLY ED STRING MATCHING:

Theorem 7.1. *We can solve the reporting or decision version of* DOUBLY ED STRING MATCHING *on an ED text T having length n_T, cardinality m_T and size N_T, and an ED pattern P having length n_P, cardinality m_P and size N_P in $\mathcal{O}(N_T m_P + N_P m_T)$ time or in $\mathcal{O}(N_T^{\omega-1} n_P + N_P^{\omega-1} n_T)$ time.*

Proof. Like in EDSI, we need to check whether an accepting state is reachable from a starting one, on the very same intersection graph. This time, however, for every state k of the path-automaton of T, we make every node $(k,1)$ reachable (starting), and every node (k, n_P+1) accepting. That way, after computing every reachable node with one of the algorithms underlying Theorem 4.6 or Theorem 4.12, an accepting node is reachable if and only if a string in $\mathcal{L}(P)$ occurs at any position of a string from $\mathcal{L}(T)$. The complexity is the same as before, since the size of graph remains unchanged.

For the reporting version, we observe that occurrences ending in set $T[i]$ correspond to reachable nodes in the set $\{(i, n_P+1)\} \cup U'_{i-1,n_P+1} \setminus \{(i-1, n_P+1)\}$. Hence, we simply report the list of such sets containing a reachable node, which can be done in $\mathcal{O}(N_T)$ total time over all i. The starting positions can be reported through the use of reversed strings. □

In [37, Corollary 10] it was shown that DOUBLY ED STRING MATCHING cannot be solved in $\mathcal{O}(N_P N_T^{1-\varepsilon})$ time or $\mathcal{O}(N_T N_P^{1-\varepsilon})$ time, for any constant $\varepsilon > 0$, unless the OV conjecture is false. This result does not contradict our result as the lengths n_P, n_T can in general be large.

Notice that if the pattern P is a standard string ($n_P = m_P = 1$) the transitions from $U'_{i,1}$ to $U'_{i,2}$ are there solely to check if P occurs in any string from $T[i]$ (which can be done in $\mathcal{O}(N_P + N_T)$ time). Hence, similarly to Corollary 4.13 the running time of our FMM algorithm is equal to $\tilde{\mathcal{O}}(n_T N_P^{\omega-1} + N_T)$. This complexity (and design) matches the result of [21].

Symmetrically, if the text T is a standard string ($n_T = m_T = 1$), then the FMM solution runs in $\tilde{\mathcal{O}}(n_P N_T^{\omega-1} + N_P)$ time (this time like in case of Corollary 4.13 the edges from $U_{1,j}$ to $U_{2,j}$ are not considered at all).

8. Approximate EDSI

Another popular extension of string equality is *approximate* equality, where one wants to find the distance between two strings: the minimal number of operations that modify one string into the other one, or decide whether this number is at most k, for a given integer $k > 0$.

We denote by $d_H(S_1, S_2)$ (resp. $d_E(S_1, S_2)$) the Hamming distance (resp. edit distance) of two standard strings S_1, S_2. The problem of finding the Hamming distance of two standard strings is easily solvable in $\mathcal{O}(N_1 + N_2)$ time, while for edit distance the time increases to $\mathcal{O}(N_1 \cdot N_2)$ time in general case [57] or $\mathcal{O}(N_1 + N_2 + k^2)$ [58] for a given upper bound k.

Approximate EDSI gives another measure of similarity when the normal intersection turns out to be empty.

Theorem 8.1. *Given a pair T_1, T_2 of ED strings of sizes N_1 and N_2, respectively, we can find the pair $S_1 \in \mathcal{L}(T_1), S_2 \in \mathcal{L}(T_2)$ minimizing the distance $d_H(S_1, S_2)$ or $d_E(S_1, S_2)$ in $\mathcal{O}(N_1 N_2)$ time.*

Proof. We adapt the classic Wagner–Fischer algorithm [57] to our case. We construct the standard (uncompacted) NFA intersection of the two path-automata for T_1 and T_2, set the weight of all its edges to 0, and then add extra edges with weight 1 that represent edit operations.

In case of Hamming distance, when a pair of transitions $u \xrightarrow{l_1} u'$, $v \xrightarrow{l_2} v'$ does not match ($l_1 \neq l_2$) we still add the weight-1 edge between (u, v) and (u', v') (l_1 is substituted with l_2).

In case of edit distance we additionally construct weight-1 edges from (u, v) to (u', v) for every transition $u \xrightarrow{l_1} u'$ (corresponding to deletion of the letter l_1), and to (u, v') for every transition $v \xrightarrow{l_2} v'$ (corresponding to insertion of letter l_2).

Now all we need to do is to find the minimum cost path from the starting node to the accepting one (in case of Hamming distance only if such a path exists), which can be done in linear time using Dial's implementation of Dijkstra's algorithm [59]. By simply spelling the labels of the found path we obtain strings S_1 and S_2 (the difference between them is encoded through the labels of the weight-1 edges). □

Without having a threshold k, even when T_1, T_2 are standard strings ($n_1 = n_2 = m_1 = m_2 = 1$), for the edit distance we cannot obtain an algorithm running in $\mathcal{O}((N_1 N_2)^{1-\varepsilon})$ time for a constant $\varepsilon > 0$ unless SETH fails [60], or a faster algorithm through a parameterization with n_1, n_2, m_1, m_2.

On the other hand, if we are given a threshold k, we can adapt another classic algorithm, given by Landau and Vishkin [61], to obtain the following theorem.

Theorem 8.2. *Given an ED string T_1 of cardinality m_1 and size N_1, an ED string T_2 of cardinality m_2 and size N_2, and an integer $k > 0$, we can check whether a pair $S_1 \in \mathcal{L}(T_1), S_2 \in \mathcal{L}(T_2)$ with $d_H(S_1, S_2) \leq k$ (resp. $d_E(S_1, S_2) \leq k$) exists in $\mathcal{O}(k(N_1 m_2 + N_2 m_1))$ time (resp. in $\mathcal{O}(k^2(N_1 m_2 + N_2 m_1))$ time) and, if that is the case, return the pair with the smallest distance.*

Proof. This time we make use of the compacted intersection automaton from Lemma 4.3. In addition to standard weight-0 edges, which are constructed when $\mathsf{LCP}(l_1, l_2) = \min(|l_1|, |l_2|)$ for l_1, l_2 being the labels of extended transitions in the two path-automata, we also add new edges when one of the strings is at distance at most k from a prefix of the other one. More formally, for a pair of extended transitions $u \xrightarrow{l_1} u'$, $v \xrightarrow{l_2} v'$, if l_1 is at distance k' from the length-x prefix of l_2, we produce a weight-k' edge from (u, v) to (u', v_x), where v_x is the implicit state between v and v' representing the length x prefix of l_2 (symmetrically for l_2 at distance at most k from a prefix of l_1). All such values of $k' \leq k$ can be found with the use of a standard LCP queries data structure and *kangaroo jumps* [62] in $\mathcal{O}(k)$ time for Hamming distance and in $\mathcal{O}(k^2)$ time for edit distance [61].

In case of Hamming distance, the total number of extended transitions is still $\mathcal{O}(N_1 m_2 + N_2 m_1)$ (the number of transition checks does not change). In case of edit distance, it can increase by a factor of k. Indeed, for a single pair of transitions l_1, l_2, we can produce up to $2k + 1$ weighted transitions.

After that, once again, we can use Dial's implementation of Dijkstra's algorithm [59] to find the smallest weight path in this graph, obtaining the claimed result. □

One may notice that the modifications to the standard intersection algorithms from this section and the previous one are independent; in the previous section those consisted of marking additional nodes as starting/accepting, while in this section those consisted of adding edges. By applying both modifications simultaneously, we can solve the Approximate (Doubly) EDSM problem, that is, the problem of finding a pair of strings $S_P \in \mathcal{L}(P), S_T \in \mathcal{L}(T)$ such that S_P is at distance at most k from a substring of S_T (for which this distance is minimized). We thus obtain the following result:

Corollary 8.3. *Approximate (Doubly) EDSM can be solved in $\mathcal{O}(N_1 N_2)$ time in general, or in $\mathcal{O}(k(N_1 m_2 + N_2 m_1))$ time for Hamming distance and $\mathcal{O}(k^2(N_1 m_2 + N_2 m_1))$ time for edit distance when we are given an integer $k > 0$ as an upper bound on the sought distance.*

Once again we can notice that when pattern P is a standard string this time complexity can be bounded by a smaller value – we can check if the pattern P appears in any of the strings from any of the sets $T[i]$ in $\mathcal{O}(k(N_P + N_T))$ total time [61]. Otherwise each edge used has at least one endpoint in a node (k, j), where j is an explicit state from the automaton representing T, hence the total number edge checks in this case is equal to $N_P m_T$ (we can compute the backwards edges the same way as the forward ones), thus we obtain the total running time of $\mathcal{O}(k(N_P m_T + N_T))$ for Hamming distance and $\mathcal{O}(k^2 N_P m_T + k N_T)$ time for edit distance. Our results match the time complexity of the corresponding results of [22]. (Note that [22] uses G to denote m_T and m to denote N_P.)

Symmetrically, when text T is a standard string those time complexities are $\mathcal{O}(k(N_T m_P + N_P))$ and $\mathcal{O}(k^2 N_T m_P + k N_P))$, respectively, since in this case every useful edge has at least one endpoint in a node (i, k), where i is an explicit state from the automaton representing P.

9. Open questions

In our view, the main open questions that stem from our work are as follows.

- We showed an $\tilde{\mathcal{O}}(n_2 N_1^{\omega-1} + n_1 N_2^{\omega-1})$-time algorithm for EDSI. Can one design an $\mathcal{O}(n_2 N_1^{\omega-1-\epsilon} + n_1 N_2^{\omega-1-\epsilon})$-time (perhaps not combinatorial) algorithm for EDSI, for some $\epsilon > 0$?
- We showed that (unless the BMM conjecture fails) there is no combinatorial $\mathcal{O}((N_1 + N_2)^{1.2-\epsilon} f(n_1, n_2))$-time algorithm for EDSI. Can one show a stronger conditional lower bound for combinatorial algorithms?
- We showed an $\mathcal{O}(N_1 \log N_1 \log n_1 + N_2 \log N_2 \log n_2)$-time algorithm for outputting a representation of the intersection language of two unary ED strings. Can one design an $o(N_1 \log N_1 \log n_1 + N_2 \log N_2 \log n_2)$-time algorithm?

CRediT authorship contribution statement

Estéban Gabory: Writing – review & editing, Writing – original draft. **Moses Njagi Mwaniki:** Software. **Nadia Pisanti:** Writing – review & editing, Writing – original draft. **Solon P. Pissis:** Writing – review & editing, Writing – original draft, Methodology, Formal analysis, Conceptualization. **Jakub Radoszewski:** Writing – review & editing, Writing – original draft, Methodology, Formal analysis. **Michelle Sweering:** Writing – review & editing, Writing – original draft, Methodology. **Wiktor Zuba:** Writing – review & editing, Writing – original draft, Software, Methodology, Formal analysis.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work was partially supported by the PANGAIA, ALPACA and NETWORKS projects that have received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreements No. 872539, 956229 and 101034253, respectively. Nadia Pisanti was partially supported by MUR PRIN 2022 YRB97K PINC and by NextGeneration EU programme PNRR ECS00000017 Tuscany Health Ecosystem. Jakub Radoszewski was supported by the Polish National Science Center, grants no. 2018/31/D/ST6/03991 and 2022/46/E/ST6/00463.

Data availability

No data was used for the research described in the article.

References

[1] D. Gusfield, Algorithms on Strings, Trees, and Sequences - Computer Science and Computational Biology, Cambridge University Press, 1997.
[2] N.R. Dixon, T.B. Martin, Automatic Speech and Speaker Recognition, John Wiley & Sons, Inc., USA, 1979.

[3] R. Baeza-Yates, B.A. Ribeiro-Neto, Modern Information Retrieval - the Concepts and Technology Behind Search, second edition, Pearson Education Ltd., Harlow, England, 2011, http://www.mir2ed.org/.
[4] P. Heckel, A technique for isolating differences between files, Commun. ACM 21 (4) (1978) 264–268, https://doi.org/10.1145/359460.359467.
[5] L.A.K. Ayad, C. Barton, S.P. Pissis, A faster and more accurate heuristic for cyclic edit distance computation, Pattern Recognit. Lett. 88 (2017) 81–87, https://doi.org/10.1016/j.patrec.2017.01.018.
[6] U. Manber, S. Wu, An algorithm for approximate membership checking with application to password security, Inf. Process. Lett. 50 (4) (1994) 191–197, https://doi.org/10.1016/0020-0190(94)00032-8.
[7] G. Navarro, A guided tour to approximate string matching, ACM Comput. Surv. 33 (1) (2001) 31–88, https://doi.org/10.1145/375360.375365.
[8] M. Crochemore, C. Hancart, T. Lecroq, Algorithms on Strings, Cambridge University Press, 2007.
[9] G. Bernardini, A. Conte, G. Gourdel, R. Grossi, G. Loukides, N. Pisanti, S.P. Pissis, G. Punzi, L. Stougie, M. Sweering, Hide and mine in strings: hardness and algorithms, in: C. Plant, H. Wang, A. Cuzzocrea, C. Zaniolo, X. Wu (Eds.), 20th IEEE International Conference on Data Mining, ICDM 2020, Sorrento, Italy, November 17-20, 2020, IEEE, 2020, pp. 924–929.
[10] IUPAC-IUB commission on biochemical nomenclature, abbreviations and symbols for nucleic acids, polynucleotides, and their constituents, Biochemistry 9 (20) (1970) 4022–4027, https://doi.org/10.1016/0022-2836(71)90319-6.
[11] M. Alzamel, L.A.K. Ayad, G. Bernardini, R. Grossi, C.S. Iliopoulos, N. Pisanti, S.P. Pissis, G. Rosone, Comparing degenerate strings, Fundam. Inform. 175 (1–4) (2020) 41–58, https://doi.org/10.3233/FI-2020-1947.
[12] E. Domingo, C. García-Crespo, C. Perales, Historical perspective on the discovery of the quasispecies concept, Annu. Rev. Virol. 8 (1) (2021) 51–72, pMID: 34586874, https://doi.org/10.1146/annurev-virology-091919-105900.
[13] C.S. Iliopoulos, R. Kundu, S.P. Pissis, Efficient pattern matching in elastic-degenerate strings, Inf. Comput. 279 (2021) 104616, https://doi.org/10.1016/j.ic.2020.104616.
[14] The Computational Pan-Genomics Consortium, Computational pan-genomics: status, promises and challenges, Brief. Bioinform. 19 (1) (2018) 118–135.
[15] V. Carletti, P. Foggia, E. Garrison, L. Greco, P. Ritrovato, M. Vento, Graph-based representations for supporting genome data analysis and visualization: opportunities and challenges, in: D. Conte, J. Ramel, P. Foggia (Eds.), Graph-Based Representations in Pattern Recognition - 12th IAPR-TC-15, International Workshop, GbRPR 2019, Tours, France, June 19-21, 2019, Proceedings, in: Lecture Notes in Computer Science, vol. 11510, Springer, 2019, pp. 237–246.
[16] J.A. Baaijens, P. Bonizzoni, C. Boucher, G.D. Vedova, Y. Pirola, R. Rizzi, J. Sirén, Computational graph pangenomics: a tutorial on data structures and their applications, Nat. Comput. 21 (1) (2022) 81–108, https://doi.org/10.1007/s11047-022-09882-6.
[17] R. Grossi, C.S. Iliopoulos, C. Liu, N. Pisanti, S.P. Pissis, A. Retha, G. Rosone, F. Vayani, L. Versari, On-line pattern matching on similar texts, in: J. Kärkkäinen, J. Radoszewski, W. Rytter (Eds.), 28th Annual Symposium on Combinatorial Pattern Matching, CPM 2017, July 4-6, 2017, Warsaw, Poland, in: LIPIcs, vol. 78, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017, pp. 9:1–9:14.
[18] A. Cisłak, S. Grabowski, J. Holub, SOPanG: online text searching over a pan-genome, Bioinformatics 34 (24) (2018) 4290–4292, https://doi.org/10.1093/bioinformatics/bty506.
[19] K. Aoyama, Y. Nakashima, T. I, S. Inenaga, H. Bannai, M. Takeda, Faster online elastic degenerate string matching, in: G. Navarro, D. Sankoff, B. Zhu (Eds.), Annual Symposium on Combinatorial Pattern Matching, CPM 2018, July 2-4, 2018 - Qingdao, China, in: LIPIcs, vol. 105, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018, pp. 9:1–9:10.
[20] G. Bernardini, P. Gawrychowski, N. Pisanti, S.P. Pissis, G. Rosone, Even faster elastic-degenerate string matching via fast matrix multiplication, in: C. Baier, I. Chatzigiannakis, P. Flocchini, S. Leonardi (Eds.), 46th International Colloquium on Automata, Languages, and Programming, ICALP, 2019, July 9-12, 2019, Patras, Greece, in: LIPIcs, vol. 132, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019, pp. 21:1–21:15.
[21] G. Bernardini, P. Gawrychowski, N. Pisanti, S.P. Pissis, G. Rosone, Elastic-degenerate string matching via fast matrix multiplication, SIAM J. Comput. 51 (3) (2022) 549–576, https://doi.org/10.1137/20M1368033.
[22] G. Bernardini, N. Pisanti, S.P. Pissis, G. Rosone, Approximate pattern matching on elastic-degenerate text, Theor. Comput. Sci. 812 (2020) 109–122, https://doi.org/10.1016/J.TCS.2019.08.012.
[23] G. Bernardini, E. Gabory, S.P. Pissis, L. Stougie, M. Sweering, W. Zuba, Elastic-degenerate string matching with 1 error, in: A. Castañeda, F. Rodríguez-Henríquez (Eds.), LATIN 2022: Theoretical Informatics - 15th Latin American Symposium, Guanajuato, Mexico, November 7-11, 2022, Proceedings, in: Lecture Notes in Computer Science, vol. 13568, Springer, 2022, pp. 20–37.
[24] R. Impagliazzo, R. Paturi, On the complexity of k-SAT, J. Comput. Syst. Sci. 62 (2) (2001) 367–375, https://doi.org/10.1006/JCSS.2000.1727.
[25] R. Impagliazzo, R. Paturi, F. Zane, Which problems have strongly exponential complexity?, J. Comput. Syst. Sci. 63 (4) (2001) 512–530, https://doi.org/10.1006/JCSS.2001.1774.
[26] R. Williams, A new algorithm for optimal 2-constraint satisfaction and its implications, Theor. Comput. Sci. 348 (2–3) (2005) 357–365, https://doi.org/10.1016/J.TCS.2005.09.023.
[27] A. Abboud, V. Vassilevska Williams, Popular conjectures imply strong lower bounds for dynamic problems, in: 55th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2014, Philadelphia, PA, USA, October 18-21, 2014, IEEE Computer Society, 2014, pp. 434–443.
[28] E. Gabory, N.M. Mwaniki, N. Pisanti, S.P. Pissis, J. Radoszewski, M. Sweering, W. Zuba, Comparing elastic-degenerate strings: algorithms, lower bounds, and applications, in: L. Bulteau, Z. Lipták (Eds.), 34th Annual Symposium on Combinatorial Pattern Matching, CPM 2023, June 26-28, 2023, Marne-la-Vallée, France, in: LIPIcs, vol. 259, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023, pp. 11:1–11:20.
[29] A. Backurs, P. Indyk, Which regular expression patterns are hard to match?, in: I. Dinur (Ed.), IEEE 57th Annual Symposium on Foundations of Computer Science, FOCS 2016, 9-11 October 2016, Hyatt Regency, New Brunswick, New Jersey, USA, IEEE Computer Society, 2016, pp. 457–466.
[30] P. Bille, I.L. Gørtz, Sparse regular expression matching, in: D.P. Woodruff (Ed.), Proceedings of the 2024 ACM-SIAM Symposium on Discrete Algorithms, SODA 2024, Alexandria, VA, USA, January 7-10, 2024, SIAM, 2024, pp. 3354–3375.
[31] M. Alzamel, L.A.K. Ayad, G. Bernardini, R. Grossi, C.S. Iliopoulos, N. Pisanti, S.P. Pissis, G. Rosone, Degenerate string comparison and applications, in: L. Parida, E. Ukkonen (Eds.), 18th International Workshop on Algorithms in Bioinformatics, WABI 2018, August 20-22, 2018, Helsinki, Finland, in: LIPIcs, vol. 113, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018, pp. 21:1–21:14.
[32] P. Gawrychowski, S. Ghazawi, G.M. Landau, On indeterminate strings matching, in: I.L. Gørtz, O. Weimann (Eds.), 31st Annual Symposium on Combinatorial Pattern Matching, CPM 2020, Copenhagen, Denmark, June 17–19, 2020, in: LIPIcs, vol. 161, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020, pp. 14:1–14:14.
[33] D. Gibney, An efficient elastic-degenerate text index? Not likely, in: C. Boucher, S.V. Thankachan (Eds.), String Processing and Information Retrieval - 27th International Symposium, SPIRE 2020, Orlando, FL, USA, October 13-15, 2020, Proceedings, in: Lecture Notes in Computer Science, Springer, 2020, pp. 76–88.
[34] M. Equi, T. Norri, J. Alanko, B. Cazaux, A.I. Tomescu, V. Mäkinen, Algorithms and complexity on indexing founder graphs, Algorithmica 85 (6) (2023) 1586–1623, https://doi.org/10.1007/S00453-022-01007-W.
[35] V. Mäkinen, B. Cazaux, M. Equi, T. Norri, A.I. Tomescu, Linear time construction of indexable founder block graphs, in: C. Kingsford, N. Pisanti (Eds.), 20th International Workshop on Algorithms in Bioinformatics, WABI 2020, September 7-9, 2020, Pisa, Italy (Virtual Conference), in: LIPIcs, vol. 172, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020, pp. 7:1–7:18.
[36] N. Rizzo, M. Equi, T. Norri, V. Mäkinen, Elastic founder graphs improved and enhanced, Theor. Comput. Sci. 982 (2024) 114269, https://doi.org/10.1016/J.TCS.2023.114269.

[37] R. Ascone, G. Bernardini, A. Conte, M. Equi, E. Gabory, R. Grossi, N. Pisanti, A unifying taxonomy of pattern matching in degenerate strings and founder graphs, in: S.P. Pissis, W. Sung (Eds.), 24th International Workshop on Algorithms in Bioinformatics, WABI 2024, September 2-4, 2024, Royal Holloway, London, United Kingdom, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2024, pp. 14:1–14:21.
[38] M. Equi, V. Mäkinen, A.I. Tomescu, R. Grossi, On the complexity of string matching for graphs, ACM Trans. Algorithms 19 (3) (2023) 21:1–21:25, https://doi.org/10.1145/3588334.
[39] C. Calabro, R. Impagliazzo, R. Paturi, The complexity of satisfiability of small depth circuits, in: J. Chen, F.V. Fomin (Eds.), Parameterized and Exact Computation, 4th International Workshop, IWPEC 2009, Copenhagen, Denmark, September 10–11, 2009, in: Lecture Notes in Computer Science, vol. 5917, Springer, 2009, pp. 75–85.
[40] V. Vassilevska Williams, R. Williams, Subcubic equivalences between path, matrix and triangle problems, in: 51st Annual IEEE Symposium on Foundations of Computer Science, FOCS 2010, October 23-26, 2010, Las Vegas, Nevada, USA, IEEE Computer Society, 2010, pp. 645–654.
[41] J.M. Kleinberg, É. Tardos, Algorithm Design, Addison-Wesley, 2006.
[42] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, Introduction to Algorithms, 3rd edition, MIT Press, 2009, http://mitpress.mit.edu/books/introduction-algorithms.
[43] M. Farach, Optimal suffix tree construction with large alphabets, in: 38th Annual Symposium on Foundations of Computer Science, FOCS 1997, Miami Beach, Florida, USA, October 19-22, 1997, IEEE Computer Society, 1997, pp. 137–143.
[44] M.A. Bender, M. Farach-Colton, The LCA problem revisited, in: G.H. Gonnet, D. Panario, A. Viola (Eds.), LATIN 2000: Theoretical Informatics, 4th Latin American Symposium, Punta del Este, Uruguay, April 10-14, 2000, Proceedings, in: Lecture Notes in Computer Science, vol. 1776, Springer, 2000, pp. 88–94.
[45] M.V. Lawson, Finite Automata, Chapman and Hall/CRC, 2004.
[46] M.L. Fredman, J. Komlós, E. Szemerédi, Storing a sparse table with O(1) worst case access time, J. ACM 31 (3) (1984) 538–544, https://doi.org/10.1145/828.1884.
[47] K. Jacobs, A. Itai, S. Wintner, Acronyms: identification, expansion and disambiguation, Ann. Math. Artif. Intell. 88 (5–6) (2020) 517–532, https://doi.org/10.1007/s10472-018-9608-8.
[48] K. Kirchhoff, A.M. Turner, Unsupervised resolution of acronyms and abbreviations in nursing notes using document-level context models, in: C. Grouin, T. Hamon, A. Névéol, P. Zweigenbaum (Eds.), Proceedings of the Seventh International Workshop on Health Text Mining and Information Analysis, Louhi@EMNLP 2016, Austin, TX, USA, November 5, 2016, Association for Computational Linguistics, 2016, pp. 52–60.
[49] D.R. Kubal, A. Nagvenkar, Effective ensembling of transformer based language models for acronyms identification, in: A.P.B. Veyseh, F. Dernoncourt, T.H. Nguyen, W. Chang, L.A. Celi (Eds.), Proceedings of the Workshop on Scientific Document Understanding Co-Located with 35th AAAI Conference on Artificial Intelligence, SDU@AAAI 2021, Virtual Event, February 9, 2021, in: CEUR Workshop Proceedings, vol. 2831, CEUR-WS.org, 2021, http://ceur-ws.org/Vol-2831/paper24.pdf.
[50] C. Kuo, M.H.T. Ling, K. Lin, C. Hsu, BIOADI: a machine learning approach to identifying abbreviations and definitions in biological literature, BMC Bioinform. 10 (S-15) (2009) 7, https://doi.org/10.1186/1471-2105-10-S15-S7.
[51] J. Liu, C. Liu, Y. Huang, Multi-granularity sequence labeling model for acronym expansion identification, Inf. Sci. 378 (2017) 462–474, https://doi.org/10.1016/j.ins.2016.06.045.
[52] A.S. Schwartz, M.A. Hearst, A simple algorithm for identifying abbreviation definitions in biomedical text, in: R.B. Altman, A.K. Dunker, L. Hunter, T.E. Klein (Eds.), Proceedings of the 8th Pacific Symposium on Biocomputing, PSB 2003, Lihue, Hawaii, USA, January 3-7, 2003, 2003, pp. 451–462, http://psb.stanford.edu/psb-online/proceedings/psb03/schwartz.pdf.
[53] K. Taghva, J. Gilbreth, Recognizing acronyms and their definitions, Int. J. Doc. Anal. Recognit. 1 (4) (1999) 191–198, https://doi.org/10.1007/s100320050018.
[54] A.P.B. Veyseh, F. Dernoncourt, Q.H. Tran, T.H. Nguyen, What does this acronym mean? Introducing a new dataset for acronym identification and disambiguation, in: D. Scott, N. Bel, C. Zong (Eds.), Proceedings of the 28th International Conference on Computational Linguistics, COLING 2020, Barcelona, Spain (Online), December 8-13, 2020, International Committee on Computational Linguistics, 2020, pp. 3285–3301.
[55] E. Gabory, N.M. Mwaniki, N. Pisanti, S.P. Pissis, J. Radoszewski, M. Sweering, W. Zuba, Pangenome comparison via ED strings, Front. Bioinform. 4 (2024), https://doi.org/10.3389/fbinf.2024.1397036, https://www.frontiersin.org/journals/bioinformatics/articles/10.3389/fbinf.2024.1397036.
[56] K. Bringmann, M. Künnemann, Quadratic conditional lower bounds for string problems and dynamic time warping, https://doi.org/10.1109/FOCS.2015.15, 2015.
[57] R.A. Wagner, M.J. Fischer, The string-to-string correction problem, J. ACM 21 (1) (1974) 168–173, https://doi.org/10.1145/321796.321811.
[58] G.M. Landau, E.W. Myers, J.P. Schmidt, Incremental string comparison, SIAM J. Comput. 27 (2) (1998) 557–582, https://doi.org/10.1137/S0097539794264810.
[59] R.B. Dial, Algorithm 360: shortest-path forest with topological ordering [H], Commun. ACM 12 (11) (1969) 632–633, https://doi.org/10.1145/363269.363610.
[60] A. Backurs, P. Indyk, Edit distance cannot be computed in strongly subquadratic time (unless SETH is false), SIAM J. Comput. 47 (3) (2018) 1087–1097, https://doi.org/10.1137/15M1053128.
[61] G.M. Landau, U. Vishkin, R. Nussinov, An efficient string matching algorithm with k differences for nucleotide and amino acid sequences, Nucleic Acids Res. 14 (1) (1986) 31–46, https://doi.org/10.1093/NAR/14.1.31.
[62] Z. Galil, R. Giancarlo, Improved string matching with k mismatches, SIGACT News 17 (4) (1986) 52–54, https://doi.org/10.1145/8307.8309.